نحو البرمجة بـXcode

ماتحتاجة لبرمجة الماكنتوش
بلغة Objective-C



كتاب من تأليف Bert Altenburg, Alex Clarke , Philippe Mougin
ترجمة : مازن الرمال

هل احببت يوماَ معرفة كيفية عمل الامور،
هل انت ممن يتابع جديد العلوم والمخترعات،
و هل انت من محبي البرمجة والتحدى؟

بين يديك كتاب قيم، كمقدمة لحقل مشوق وجدت فيه ما يفيدني، واحببت مشاركتكم به، ارجو الا تبخلوا على بمرئياتكم وملاحظاتكم على البريد
maxenco@mac.com

نحو البرمجة بـXcode

ماتحتاجة لبرمجة الماكنتوش
بلغة Objective-C

كتاب من تأليف
Bert Altenburg
Alex Clarke
Philippe Mougin

ترجمة : مازن الرمال

النسخة ١,١

ترخيص

المحتويات

المقدمة

تقدم لك تكنولوجيا أپل Apple كافة الأدوات اللازمة لإنشاء تطبيقات Cocoa مجاناَ. فهذه الأدوات تأتيك مجتمعة تحت مظلة Xcode التي توجد بنظام Mac OS X كما انها قابلة للتحميل من قسم المطورين بموقع أپل الألكتروني.

يوجد عدد من الكتب الجيدة حول برمجة الماكنتوش، ولكنها تفترض وجود خبرة برمجيه لدى القارئ. هذا الكتاب لن يفترض ذلك. وسيعلمك أساسيات البرمجة بلغة Objective-C مستخدماَ برنامج Xcode.

بعد قرأتك لعدد من فصول الكتاب ستتمكن من إنشاء برنامج بسيط لا يعتمد استخدام واجهة التطبيقات الرسوميه – Graphical User Interface والمعروفة باختصارها الشائع GUI . وبعد ذلك بعدد من الفصول ستتعلم إنشاء برنامج بسيط من خلال واجهة التطبيقات الرسوميه (GUI). وبعد انتهائك من قراءة هذا الكتاب ستكون جاهزاَ لما تقدمة لك الكتب المتقدمة لما ورد بعاليه. عليك بتفحصها وقراءتها حيث يوجد الكثير لتتعلمه. حالياً لا داعي للقلق فهذا الكتاب ينتهج أسلوب التقديم السهل للمعلومات.

كيف تستخدم هذا الكتاب

كما سترى، سنستعرض بعض النصوص التي تمثل الشفرة البرمجية Code snippest التي سيقوم الماكنتوش بتقييمها وتنفيذ محتواها وستكون داخل صناديق مثل هذا:

```
Some tidbits
```

ونقترح عليك أن تقرأ الفصل الواحد مرتان على الأقل مع تجاهل صناديق الشفرة البرمجية في البداية. ثم أثناء قراءتك الثانية للفصل إقراء هذه الصناديق، حيث ستكون عندها متأهباً لإعادة تذكر وتطبيق ما تعلمته فغالباً ما تكون هذه النصوص مشتتة لانتباهك بالمرات الأولى. إن اعتماد هذه الطريقة في تعلم مبادئ جديدة يعينك من تثبيت المعلومة.

هذه الكتاب يحتوي عدة أمثلة مكونة من سطر أو اثنين من الشفرة البرمجية. وقد استخدمنا الأقواس التالية[4] لعمل ارتباط للمعلومة مع مثالها المقابل. إن غالبية الأمثلة مكونة من سطرين أو اكثر قليلاً من الشفرة البرمجية. وفي أحيان معينة سيتضمن المثال رقم ويتبعه رقم آخر بعد الفاصلة حيث نستخدم الرقم الآخر للإشارة إلى رقم سطر معين من كتلة شفرة البرمجة. كمثال القوس [3.4] يشير إلى السطر ٣ من المثال٤ . وفي الاقتباسات الطويلة من الشفرة سنقوم بالإشارة إلى سطر معين كما يلي:

```
volume = baseArea * height // [4.3]
```

في الحقيقة إن عملية البرمجة ليست بالأمر اليسير. فهي تتطلب منك الحضور الواعي والتجريب التطبيقي لكافة المعلومات التي يقدمها هذا الكتاب. لن تستطيع العزف على البيانو أو قيادة السيارة من خلال قراءة الكتب. الأمر ذاته ينطبق مع البرمجة.

هذا الكتاب ُمقدم لك بهيئتة الإلكترونية لتمكينك من الانتقال منه إلى برنامج Xcode والعودة دون أية عوائق. لذا ما أن تبدأ مع الفصل الخامس نقترح عليك المرور بكل فصل ثلاث مرات على الأقل. في المرات اللاحقة حاول تطبيق الأمثلة الواردة ولا مانع من أن تجري تعديلاتك على الشفرة البرمجية حتى تجرب كيفية سير الأمور.

قبل أن نبدأ

لقد قمنا بكتابة هذا الكتاب وكنت عزيزي القاريء محور اهتمامنا . فبما أنه مجاني، أسمح لنا بتسويق الماكنتوش كأفضل نظام تشغيل بقليل من الجهد . وإليك الطريقة :

١ . كلما صقلت خبرات تعاملك مع الماكنتوش كلما سهّلت للمحيطين بك أن يتنبهوا لإمكانيات هذا النظام . لذا كن مطلعاَ على آخر الأخبار بزيارتك للمواقع والمنتديات وقراءتك للمجلات التي تهتم بالماكنتوش . من المؤكد أن تعلمك للغة Objective-C أو إتقانك لـ AppleScript سيكون لهما اثرهما الفاعل في الأعمال التي تؤديها .

إن استخدام AppleScript يغنيك عن ساعات من الأعمال المتكررة والمضنية . ألقي نظرة على كتابنا المجاني حول AppleScript والموجه للمبتدئين وهو متوفر على http://www.macscripter.net/books .

٢ . اظهر للعالم كافة أن الحاسبات ليست فقط "وندوز Windows" . إن لبس ملابس عليها شعارات ماكنتوش للعلن هو أحد الطرق، ولكن هناك طرق اكثر لتعزيز هذا النظام وهي تبدأ من داخل بيتك .

فإذا ما شغلت مراقب النشاطات **Activity Monitor** الموجود بمجلد الأدوات **Utilities** المتواجد داخل مجلد التطبيقات **Applications** ستلحظ مدى ندرة استخدام النظام لكافة قواه الحسابية القصوى. يقوم العلماء بتطوير عدة مشاريع حوسبة ذات تحكم بعدي **distributed computing (DC)** كمشروع **Folding@home** أو مشروع **SETI@home** الذي يستقي جزء بسيط لا يذكر من موارد العمليات وهي مشاريع ذات أهداف سامية. حيث يمكنك تحميل أحد تلك البرامج المجانية وهي تدعى عميل **DC client** ودعها تنجز اعماك الحساب المطلوبة منها. انها تفّعل ذاتها ضمن النظام بأدنى قدر من طاقة المعالجة المتاحة.

ما أن تستخدم أحد برامجك الاعتيادية على جهازك سيتطلب الأخير الانتباه الأقصى من الموارد، عندها يتنحى برنامج العميل حتى أنك لن تشعر به وهو في الحدود الدنيا من نطاق النظام.

كيف لهذه المشاركة أن تساعد من وضع التوعية بالماكنتوش؟

في الحقيقة كثير من برامج مشاريع العميل **DC client** هذه تقوم بتقييد نتائج معطياتها في مواقع ألكترونية حول الوحدات التي تم احتسابها. ما أن تنضم لفريق الماكنتوش (والذين ستتعرف على أسمائهم من خلال النتائج المعروضة) عندها ستجد نسبة مستخدمي الماكنتوش والذي سيكون لك دور مهم في رفع نسبة مشاركتهم. عندها سيدرك مستخدمي الأنظمة الأخرى كيف هو أداء الماكنتوش.

يوجد هناك برامج عملاء بعدية تغطي مواضيع متنوعة كالحساب، والعلاجات السريريه والمزيد. حتى تختار المشروع المناسب تفحص هذا الموقع **http://distributedcomputing.info/projects.html** وقد توجد مشكلة بسيطة حول هذا الاقتراح: هذه المشاركة ستتحول إلى نوع من الإدمان!

٣. تأكد من حصول جهازك على أفضل البرمجيات. ولا تعتمد في ذلك على ما تطوره من برمجيات بنفسك. واجعلها عادة من عاداتك الدائمة أن تزود مطوري البرامج التي تستخدمها بمرئياتك. حتى في حال تجريبك لاحد البرامج التي لم تعجب بها، فقط أخبر المطور بالسبب. اعلن عن مشكلات البرنامج **bugs** من خلال التوثيق الدقيق للأوامر التي قُمت بها أو المحتملة لظهور تلك المشكلة.

٤. ادفع قيمة البرامج التي تستخدمها. ولتعلم انه طالما كان هناك طلبات شراء لسوق برامج الماكنتوش فالفرصة كبيرة لظهور مطورين وبرامج قيّمة.

٥ . قم بالتواصل على الأقل مع ثلاثة مستخدمين للماكنتوش على أن يكونوا متحمسين للبرمجة . أخبرهم عن هذا الكتاب، وكيف يمكنهم الحصول عليه . أو افعل خيراَ بإِخبارهم عن أهمية النقاط الأربعة السابقة .

والآن أثناء قيامك بتحميل أحد تلك البرامج البُعدية، دعنا نبدأ العمل!

البرنامج عبارة عن تعليمات متتالية

عند تعلمك قيادة السيارة، وجب عليك إن تتعامل مع عدة أمور في وقت واحد. حيث توجب عليك حينها معرفة دور مبدل الكلتش ودواسة الوقود و المكابح. فالبرمجة كذلك تتطلب منك معرفة أمور عديدة، والا تعرض برنامجك للتحطم.

فبالرغم من إن محتويات السيارة مألوفة لديك إلا أن ذلك لا يعد ميزة أثناء التعامل مع **Xcode**. وحتى لا نثقل عليك، تركنا المسائل البرمجية لآخر هذا الفصل. أولا سنجعلك تتعامل بارتياح مع شفرة **Objective-C** وذلك من خلال عرض أساسيات حسابية مألوفة لديك. ففي التعليم الإبتدائي وجب عليك القيام ببعض المعاملات الحسابية مثل ملء الفراغ

```
٦+٢=... فراغ   أو   فراغ ..... = ٣*٤
```

(النجمة هنا تحل محل معامل الضرب) وفي التعليم المتوسط، تحولت نقاط الفراغات إلى نوعاَ منقرضاً من العمليات وتم الاستعاضة عنها بـ س و ص (واطلق عليها موضة الجبر) حيث

كانت تلك هي الصيحة السائدة بتلك الفترة. وقد تتساءل عن أحوال أولئك الذين فقدوا اهتمامهم لتقديم الحلول بسبب تغيير طفيف في التسمية "المتغيرات Variables".

```
٦+٢=س
ص=٣*٤
```

إن لغة Objective-C تستخدم المتغيرات كذلك. فالمتغيرات ما هي إلا أسماء تشير لجزئيات معينة من البيانات، كالأعداد مثلاً. هنا لدينا مثال [1] يعّد كتصريح لإيعاز Objective-C حيث أوجدنا متغيراً اعطيناه اسماً واسندنا له قيمة عددية من خلال سطر برمجي.

```
[1]
x = 4;
```

إن المتغير x يحتوي قيمة عددية تساوي ٤. وستلاحظ وجود فاصلة منقوطة ";" في نهاية الإيعاز. ذلك أن الفاصلة المنقوطة مطلب أساسي عند نهاية كل إيعاز. لماذا؟

قد يبدو المثال غير ذي أهمية، على كل حال الحاسبات إجمالا لا تدرك ما عليها أن تفعله بهذا السطر. لذا يوجد برنامج خاص يطلق عليه اسم "المجّمع compiler" وهو ضروري لتحويل نص التعليمة تلك إلى إيعازات و أوامر مكونة من نبضات الصفر والواحد حيث يستطيع الحاسب عندها القيام بعمله.

إن قراءة واستيعاب النصوص المكتوبة من قبل البشر مهمة صعبة للمركم، لذا ينبغي عطاءه تلميحات خاصة، كتلميحة إنتهاء سطر المعلومة من الاوامر والبيانات مثلاَ حيث وضعنا الفاصلة المنقوطة.

ان مجرد اغفال فاصلة منقوطة واحدة لاحد اسطر الشفرة البرمجية، يسبب فشل في مهمة المجّمع، ذلك انه سيفشل في تحويل تلك التعليمات الى اوامر يمكن للماكنتوش ان يفسرها وينفذها. لا تدع هذا الامر يقلقك كثيراَ، ذلك ان المجّمع سيشكو من عدم تمكنه من اتمام عملية التفسير. كما سنرى في التفصول القادمة، انه يقدم لك العون للبحث عن الاخطاء المحتملة بين سطور شفرة التعليمات.

بالرغم من ان اسماء المتغيرات ليست ذات معنى قيّم للمركم، الا ان اسماء المتغيرات الواضحة تجعل من قراءتك للبرنامج ايسر للفهم. ان هذه الخاصية مهمة جداَ عند تقصيك عن الاخطاء البرمجية داخل سطور الشفرة.

يُطلق على الاخطاء البرمجية تعارفاَ مصطلح Bug. بينما تقصي واصلاح تلك الاخطاء يطلق عليه debugging. لذا، متى ما تعاملت مع الشفرة تجنب استخدام تلك المسميات الغير واضحة

أو التي لا معنى لها مثل استخدام متغير x. مثلا عند الحاجة لوجود متغير مهمته حفظ قيمة عرض صورة ما، حيث يمكننا ان نطلق عليه اسم **pictureWidth** كالمثال **[2]** .

```
[2]
pictureWidth = ٨;
```

من منطلق ما يحّدثه المجّمع من توقف وفشل كبير لمجرد نسيانك لإدراج الفاصلة المنقوطة عند نهاية السطر، سيجعلك ذلك تدرك أن البرمجة تهتم بالتفاصيل. أحد تلك التفاصيل الواجب التنبه لها هي حقيقة حساسية النص **case-sensitive**. ذلك أن استخدام حروف لاتينية كبرى لتعريف متغير يؤدي لأخطاء فيما لو استخدمت الحروف الصغرى للإشارة للمتغير المقصود. فاسم المتغير **pictureWidth** ليس هو ذاته المتغير **pictureWIDTH** أو المتغير **PictureWidth**.

وكأسلوب عام تم الاتفاق على منهج عام لتسمية المتغيرات اتُفق على أن تكون تسميات المتغيرات من خلال وضع الحروف الكبرى في عدة مواقع من تعريف اسم المتغير، فالحرف الأول من الكلمة الأولى صغير وما يليه من كلمات تالية يكون أول حرف منها كبير، كما هو موضح بالمثال **[2]**.

وعند الالتزام بهذا الأسلوب يمكنك أن تتلافي أخطاء البرمجة بشكل كبير فيما يتعلق بمسألة حساسية الحروف.

الرجاء ملاحظة تسمية المتغيرات بكلمات متواصلة تكون في النهاية كلمة واحدة. فرغم وجود حرية كبيرة في اختيار أسماء المتغيرات، إلا أن هناك قواعد مهمة يفترض بالتسمية ان تتطابق معها.

وبالرغم من قدرتي على سردها كامله، الا ان ذلك سيدعو للملل. على كل حال القاعدة الاساسية هنا تشترط الا يكون اسم المتغير محجوز للمفردات المحجوزة للغة Objective-C (اي ان لا يكون الاسم ذي دلالة معينة لدى Objective-C).

ان تجميع اسم المتغير من عدة كلمات كالمتغير **pictureWidth** يجعل طريق برمجتك آمن من الاخطاء. ولجعل المتغير قابل للقراءة يفضل دوماَ استخدام صيغة الحروف الكبرى كفاصل للكلمات داخل اسم المتغير. ان التزامك بهذا التوجه يعصمك من اخطاء برمجية كثيرة.

في حال اصرارك لتعلم اكثر من قاعدة لتسمية المتغيرات، أكمل القراءة فبالاضافة الى الحروف يمكنك استخدام الاعداد شريطة الا تكون في بداية اسم المتغير. كذلك يمكنك استخدام الساطرة السفلية "_" للبدأ باسم المتغير أو الفصل بين الكلمات المكونه له.

هناك عدة طرق لاطلاق الاسماء، فعلى سبيل المثال:

- اسماء متغيرات مقبولة : door8k, do8or, do_or
- اسماء متغيرات غير مقبولة : door 8 (تحوي فراغ) , 8door (تبدأ برقم).
- لا ينصح باستخدامها : Door8 (بداية الاسم حرف كبير)

والان بعد ان تعلمنا كيف نسند قيمة للمتغير، يمكننا اجراء بعض الحسابات، لنتفحص الشفرة البرمجية التي تقوم بإحتساب مساحة سطح الصورة. بالمثال [3].

```
[3]
pictureWidth=8;
pictureHeight=6;
pictureSurfaceArea=pictureWidth*pictureHeig
ht;
```

سترى كيف ان المجّمع لا يصدر اي اعتراض عند وجود فراغات(المسافات) (عدا التي توجد في اسماء المتغيرات أو الكلمات المحجوزة،، الخ.) ولنجعل الشفرة البرمجية ايسر قرائة، يمكننا اضافة الفراغات كالمثال [4].

```
[4]
pictureWidth = 8;
pictureHeight = 6;
pictureSurfaceArea = pictureWidth *
pictureHeight;
```

والان تفحص المثال [5] ، وخصوصاَ السطران الاولان.

```
[5]
pictureWidth = 8;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth *
pictureHeight;
```

يمكنا تصنيف الاعداد عموماَ الى نوعان : صحيحة integers (اعداد كاملة مثل {١ و ٢٣ و ٤٥٦ و ٩٩٩}) واعداد كسرية fractional مثل {١.٥ و ٧.٢٥}. كما ترى في المثال [5.1] و [5.2] سنستخدم الاعداد الصحيحة لاجراء العمليات الحسابية وسنستخدمها احياناَ اخرى للعمليات المتكررة لعدد معين من المرات انظر الفصل ٧. وسنستخدم الاعداد الكسرية أو ما يطلق عليها floating-point كقيم حسابية لاحتساب متوسط اهداف كرة السلة مثلاَ.

ان الشفرة البرمجية بالمثال [5] غير قابلة للتنفيذ. ذلك ان المجّمع يشترط عليك اخباره مسبقاَ باسماء المتغيرات التي ستسخدمها

بالبرنامج، وان توضح له نوع البيانات التي ترغب باسناد القيم اليها، فهل هي صحيحة ام كسرية. تقنياً ذلك ما يطلق عليه التصريح عن المتغير **"declare a variable"**

```
[6]
int pictureWidth;
float pictureHeight, pictureSurfaceArea;
pictureWidth = 8;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth *
pictureHeight;
```

في السطر **[6.1]** يدلنا **int** على ان تصريح نوع المتغير **pictureWidth** او من نوع عددي صحيح وفي السطر التالي نصّرح عن متغيران دفعة واحدة. مع فصل المتغيرات بواسطة الفاصلة. بالاضافة الى ذلك نرى الأيعاز بالسطر **[6.2]**يصّرح عن ان كلا المتغيران مصنفان كمتغيرات كسرية، اي ارقام ذات جزء كسري.

في هذه الحالة من السذاجة ان يكون المتغير **pictureWidth** من نوع تصنيفات اخرى غير العددية (الصحيحة منها أو الكسرية). على كل حال سترى ان مجموع ضرب الاعداد الصحيحة التصنيف مع انواع كسرية التصنيف ينتج عنه ناتج من نوع كسري، وذلك هو سبب تصويحنا عن نوع المتغير **pictureSurfaceArea** كمتغير كسري بالسطر **[6.2]**.

لماذا قد يرغب المجّمع بمعرفة نوع مصنف المتغير سواء كان صحيح أو كسري؟

ذلك ان برامج الحاسبات تستخدم جزء من الذاكرة. والمجّمع مسئول عن حجز حيز الذاكرة المناسب (بالبايت) لكل متغير يصادفه، وبما ان هناك مصنفات وانواع متعددة للبيانات مثل ما رأينا من صحيحة وكسرية، فان كل نوع يستهلك قيمة مختلفة عن الآخر اثناء حجز الذاكرة أو اثناء العرض، فالمجّمع بحاجة لمعرفة تلك القيم حتى يتمكن من القيام بالحجز والاظهار الصحيح لكافة المصنفات.

ماذا لو اننا نتعامل مع اعداد خيالية أو اعداد ذات دقة عالية؟ لن يسعها ان تُحتوى ضمن تلك المساحات البسيطة المحجوزة من قبل المجّمع، اليس كذلك؟

نعم ذلك صحيح. وتوجد اجابتان لهذا السؤال: الاجابة الاولى تقول، ان كلا من الاعداد الصحيحة والاعداد الكسرية لديهما ظهيرين يمكّناهما من حفظ اعداد الاكبر (او تلك اعداد ذات الدقة العالية).

فاغلب انظمة التشغيل تعتمد أنواعاَ من اصناف الاعداد التخيلية الكبيرة كنوع أعداد **long long** المستخدم للاعداد الصحيحة ونوع **double** المستخدم للاعداد الكسرية. ولكن حتى تلك

يمكنها ان تُغرق حيزها من الملء، مما يقودنا للاجابة ذات الشق الثاني : حيث انه من واجبك كمبرمج ان تتنبه لمثل هذه المشاكل. على كل حال، هذه المشكلة غير مناسبه لان تناقش في الفصل الاول من كتاب للمبتدئين.

للعلم يمكن لقيمة كلا النوعين (الصحيح والكسري) ان تكون سالبة، كحسابك البنكي مثلاَ. في حال يقينك من ان القيمة لن تكون سالبة، يمكنك عندها زيادة المدى المقبول للقيمة المتاحة.

```
[7]
unsigned int chocolateBarsInStock;
```

انت تعلم يقيناَ انه لن يوجد لديك لوح شوكولاته بقيمة سالبة، لذا يمكنك استخدام تنويه **unsigned int** الموضح بالمثال [7] . ان الارقام الغير سالبة يمكنها ان تكون بقيمة اكبر أو تساوي صفر.

يمكنك دوماَ الاعلان عن المتغير واسناد قيمته دفعة واحدة حيث يمكن لهذا السطر توفير بضعة اسطر من الاسنادات. كما هو موضح بمثال**[8]** .

```
[8]
int x = 10;
float y= 3.5, z = 42;
```

في المثال السابق، قمنا باجراء عملية ضرب مستخدمين الرموز التالية، وهي تعرف رسمياَ باسم المُعامِلات الحسابية لانها تقوم باجراء العمليات الحسابية الاساسية.

```
+ معامل الجمع
- معامل الطرح
/ معامل القسمة
* معامل الضرب
```

باستخدام تلك المعاملات، يمكننا اجراء عمليات حسابية متنوعة. في حال القيت نظرة على شفرة برمجية باحد كتب برمجة **Objective-C**،سيصادفك ترميز عجيب، وغالباَ ان كاتب الشفرة كاتب كسول. فبدلا من كتابة

```
x = x + 1;
```

يميل المبرمجون لاستخدام شيء من هذا القبيل مثل **[9]** أو **[10]**

```
[9]
x++;
```

```
[10]
++x;
```

في كلتا الحالتين ذلك الايعاز يعني : قم بزيادة قيمة x بمقدار ١. في حالات خاصة يجدر الانتباه لموضع ++ سواء قبل ام بعد المتغير . تفحص المثالان التاليان **[11]** و **[12]**

```
[11]
x = 10;
y = 2 * (x++);
```

```
[12]
x = 10;
y = 2 * (++x);
```

ففي المثال [11] وبعد كل ما قيل، ستكون قيمة y مساوية لـ ٢٠ بينما x تساوي ١١. على النقيض من ذلك في الايعاز [12.2], ،سيتم اضافة قيمة x بمقدار ١ قبل اجراء عملية الضرب بـ ٢، لذا في النهاية ستكون x بقيمة ١١ و قيمة y مساوية لـ ٢٢. ان الشفرة البرمجية في المثال [12] مطابقة للمثال [13].

```
[13]
x = 10;
x++;
y = 2 * x;
```

من الواضح ان المبرمج قد قام بدمج عمليتين حسابيتين في خطوة واحدة. وشخصياَ اعتقد ان هذا الاسلوب يزيد من صعوبة قراءة وفهم البرنامج. في حال اعتمادك لتلك المختصرات لا بأس بذلك، ولكن تنبه لوجود اخطاء برمجية محتملة هنا وهناك.

قد تكون المعلومات التالية قديمة، ولكن الاقواس تحدد ترتيب إجراء العمليات الحسابية. فمن الطبيعي ان يتم اجراء عمليات الضرب * والقسمة / اولاَ قبل عمليات الجمع والطرح.

ان تاتج ٢*٣+٤ سيكون ١٠. أما باستخدام الاقواس فإننا نقوم بإجبار تنفيذ العمليات الدنيا قبل العمليات العليا ليكون ناتج عملية ٢*(٣+٤) يساوي ١٤.

أما القسمة فإنها عملية تستدعي بعض الانتباه، سبب ذلك يعود للاختلاف الاساسي عند اجراءها على اعداد صحيحة أو اعداد كسرية. تفحص المثالين [14, 15] التاليين:

```
[14]
int x = 5, y = 12, ratio;
ratio =  y / x;
```

```
[15]
float x = 5, y = 12, ratio;
ratio =  y / x;
```

في المثال الاول كانت القيمة الناتجة تساوي ٢. بينما في المثال الثاني ستكون القيمة الناتجة هي التي توقعتها اي ٢٫٤

لاتهمل التعليقات ! !

نستطيع باستخدام أسماء ذات دلالة للمتغيرات ان نجعل من الشفرة البرمجية نصوصاَ قابلة للقراءة والتفسير المنطقي[1] .

```
[1]
float pictureWidth, pictureHeight,
pictureSurfaceArea;
pictureWidth = 8.0;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth *
pictureHeight;
```

وبالرغم من اننا ملتزمين بعرض امثلة بعدد محدود من الاسطر، الا ان البرامج الكاملة التى تؤدي أعمال ذات قيمة غالباَ ما تكون شفرتها البرمجية الموصّفة اطول من ذلك بكثير.

ان توصيف وتوثيق شفرة البرنامج عملية بالغة الاهمية، فهي تمثل الهم الاكبر من مجرد الحرص على ان البرنامج يعمل بالشكل صحيح . فبعد فترة من الزمن قد ترغب بتغيير جزئية من الشفرة، عندها ستجد ان هذه التعليقات والملاحظات تلعب دوراَ مهما لتوثيق واستيعاب مهمة هذا الجزء من الشفرة، ولماذا اوجدت – الشفرة– بالمقام الاول.

نصحك بجدية ان توثق شفرتك البرمجية، انها استثمار مستقبلي. فبالاضافة الى ذلك تلعب عملية التوثيق دورها الرئيس متى ما قمت بمشاركة شفرتك البرمجية مع مبرمجين اخرين. ولعمل ملحوظة أو تعليق ضعها بعد سطرين مائلين.

```
// This is a comment
```

يُظهر **Xcode** التعليقات باللون الاخضر. متى ما كان التعليق طويل ويمتد لاكثر من سطر، ضعه ما بين */ و/*

```
/* This is a comment
extending over two lines */
```

سنتطرق لعمليات تقصي اخطاء البرنامج قريباً، حيث يوفر لك **Xcode** عدد من الامكانيات الفريدة. وان احد الطرق الاقدم في التقصي يطلق عليها اللاتعليقات **outcommenting** وذلك من خلال ادراج جزء من الشفرة البرمجية بين تلك الاسطر المائلة حتى ترى كيف تعمل بقية اوامر الشفرة.

هذا الاسلوب يمكّنك من تقصي الاخطاء البرمجية متى ما كان الجزء المحجوب متسبباً في تلك الاخطاء، كأن تحجب قيمة احد المتغيرات، حيث يمكنك حجبها مؤقتاً، ثم اسناد قيمة اختبارية لتجربة اداء بقية اجزاء الشفرة البرمجية مع تلك القيمة الاختبارية.

لا يمكنك التقليل من قيمة التعليقات، فهيا مفيدة جداَ لشرح تداخل خطوات البرنامج بلغة صريحة وواضحة. ان شرح ما تقوم به الشفرة البرمجية يمّكن المبرمجين من تحديد الجزء الاولى بالتفحص والمعاينة.

يجدر بك استخدام التعليقات لشرح الاوامر والتعليمات الاجرائية التي توّصف وتشرح البرنامج، حيث يستحيل بدونها فهم وتفسير دور بعض تلك الاجزاء من البرنامج عند الاعتماد على الشفرة فقط.

كمثال لبرنامج حسابي يستخدم معامل رياضي مبني من خلال وصف تفصيلي وارد بأحد الكتب، يجدر بك التنويه عن ذلك الارتباط بتعليقات داخل شفرتك البرمجية. وفي بعض الاحيان يفضل القيام بعمل كتابة التعليقات قبل البدء الفعلي بكتابة الشفرة البرمجية. تلك التعليقات تعينك اثناء بناء البرنامج وستكون نتائج المعاينة افضل وايسر.

ان امثلة الكتاب ليست بالضرورة موثقة كما هي الحال دائماَ، ذلك انها ترد إليك وهي محاصرة بالشرح والتوضيح.

الدالات الوظيفية

ان اطول شفرة رأيتها حتى الان مكونة من خمسة اسطر. كثير من البرامج مكونة من عدة الاف من الاسطر و قرائتها تكون مدعاة للملل، دعنا نناقش طبيعة لغة Objective-C في ترتيبها لعناصر البرنامج بشكل مبكر.فمتى ما كان البرنامج مكون من عدة الاف من الاسطر والاوامر المتلاحقة، تكون نسبة تقصي الاخطاء البرمجية اكثر صعوبة.

إن ذلك سيؤدي لوجود سلسلة متواصلة من الاوامر المتماثلة و المتكررة بحيث تتشابك بعدة اماكن من البرنامج. وهذا التشابك بين الاوامر اشبة ما يكون بصحن معكرونة **spaghetti code**. ومتى ماكان هناك خطاء برمجي في احد تلك السلاسل، سيتطلب اصلاحها تكرار عملية الاصلاح باي موقع تكرر ورودها فيه،. انه الكابوس بعينة! فمن السهل جداَ ان يُنسى تعديل جزء أو اثنان اثناء عملية التصحيح!

لذا فكر المبرمجون بعدة طرق لتنظيم الاجزاء المكونة للبرنامج، مما سهل من عمليات تقصي الاخطاء البرمجية. لقد تمثل الحل بحصر جميع تلك المجموعات من الاوامر والاجراءات بناء على طبيعة ادائها.

كمثال يمكنك جمع عدة اوامر تمكّنك من حساب مساحة سطح الدائرة. وما ان تتأكد من مدى صلاحية وصحة عمل تلك المجموعة من الاوامر ونتاجها الصحيح، فانك لن تعود اليها مرة اخرى لتقصي الاخطاء.

ان مجموعة الاوامر تلك هي ما يطلق عليها الدوال / الدالات البرمجية **function** ويمكننا اطلاق مسميات مميزة لها، حيث يمكننا استدعاء وظائفها من خلال ايراد اسمائها داخل الشفرة البرمجية.

هذا المبدء كان بمثابة التوجه الرئيس لاستخدام الدوال، حتى ان اي برنامج يتكون من دالة واحدة على الاقل تعرف باسم **main()**. هذه الدالة **main()** هي ما يبحث عنه المجّمع، انه يستدل بها ليبداء تنفيذ الاوامر والاجراءات التالية وقت تشغيل وتنفيذ البرنامج. دعنا نتفحص الدالة **main()** بمزيد من التفصيل[1].

```
[1]
main()
{
  // Body of the main() function. Put your
code here.
}
```

البيان [1.1] يظهر اسم الدالة "**main**" يتبعة اقواس. ان كلمة **main** كلمة محجوزة، حيث تمثل دالة مطلوبه اساساً لان يكون هناك برنامج، ويمكن لدالاتك الاخرى ان تُسمى باي اسم اخر. لاحظ الاقواس هنا، انها موجودة لسبب مهم جداَ، الا اننا سنناقش ذلك لاحقاَ خلال هذا الفصل.

في الاسطر التالية من الشفرة سنجد اقواس معقوفة. حيث يُتوقع منا ادراج الشفرة البرمجية أو ما يطلق عليه البنية البرمجية للدالة **body**. لقد نسخنا الشفرة البرمجية من الفصل السابق ووضعناها في مكانها الصحيح حيث يجب ان تكون[2].

```
[2]
main()
{
  // Variables are declared below
  float pictureWidth, pictureHeight,
pictureSurfaceArea;

  // We initialize the variables (we give
the variables a value)
  pictureWidth = 8.0;
  pictureHeight = 4.5;

  // Here the actual calculation is
performed
  pictureSurfaceArea = pictureWidth *
pictureHeight;
}
```

اذا ما استمرينا بادراج مزيد من اسطر الشفرة البرمجية لبنية الدالة **main()** سنصل لنتيجة حتمية مفادها استحالة أو صعوبة اجراء اي تقصي للأخطاء.

ذلك ان تتالي تلك الاسطر يقودنا تجاه بنية غير مهيكلة **unstructured** يجدر بنا تلافيها اثناء البرمجة. لذا دعنا نعيد كتابة البرنامج وذلك باستخدام الهيكلة **structure**. بغض النظر عن الدالة الاجبارية، يمكننا استحداث دالة جديدة ونطلق عليها اسم **circleArea()** كما بالمثال [3] التالي:

```
[3]
main()
{
  float pictureWidth, pictureHeight,
pictureSurfaceArea;
  pictureWidth = 8.0;
  pictureHeight = 4.5;
  pictureSurfaceArea = pictureWidth *
pictureHeight;
}

circleArea()   // [3.9]
{

}
```

لقد كان ذلك سهلاَ، الا ان دالتنا **circleArea()** المستحدثة لا تقوم باي شيء حالياَ. لكن لاحظ ان توصيف الدالة **circleArea()** وبنيتها البرمجية خارج نطاق الدالة الرئيسية **main()**، بمعنى اخر هذه الدالة غير معششة بالدالة الرئيسة.

وحتى تؤدي دالتنا الجديدة **circleArea()** دورها يجب علينا استدعائها داخل نطاق الدالة الرئيسة. لنرى كيف يتم ذلك.

```
[4]
main()
{
  float pictureWidth, pictureHeight,
pictureSurfaceArea,
        circleRadius, circleSurfaceArea;
// [4.4]
  pictureWidth  = 8.0;
  pictureHeight = 4.5;
  circleRadius  = 5.0; // [4.7]
  pictureSurfaceArea =  pictureWidth *
pictureHeight;

  // Here we call our function!
  circleSurfaceArea = circleArea(circleRadi
us); // [4.11]
}
```

ملحوظة: باقي شفرة البرنامج غير مكتملة هنا انظر المثال [3] .

لقد اضفنا متغيران بمسميات من نوع عددي كسري **float** في[4.4]،

وقد قمنا بتهئية المتغير initialize - بمعنى اسندنا له قيمة مبدئية [4.7]– ومن اكثر ما يلفت النظر [4.11] كيفية استدعاء الدالة circleArea() فكما ترى لقد ادرجنا اسم المتغيرcircleRadius بين اقواس فهو معامل للدالة circleArea() .

سوف يتم تمرير قيمة المتغير circleRadius الى الدالة circleArea() . ومتى ما قامت الدالة بالانتهاء من انجاز خطوات عملها حسابياَ ستقوم بارجاع نتيجة حتمية .دعنا نقوم بتعديل الدالة من المثال [3] لتكون كما هي بالمثال[5] .و سنعرض دالة circleArea() فقط .

```
[5]
circleArea(float theRadius) // [5.1]
{
  float theArea;
  theArea = 3.1416 * theRadius * theRadius;
// pi times r square   [5.4]
  return theArea;
}
```

بالسطر [5.1] صرحنا بان تكون قيمة معامل دالة circleArea() قيمة كسرية. متى تسلم المعامل قيمته، ستخزن بالمتغير المسمى theRadius.

ولقد استخدمنا متغيراَ اخر –استخدمنا theArea لتخزين ناتج العملية الحسابية [5.4]، لذا يجب علينا التصريح عنه [5.3]، وذلك بنفس الاسلوب المستخدم للتصريح عن الدالة main [4.4] .

حيث ستلحظ ان التصريح عن المتغير theRadius قد تم خلال اقواس [5.1] . ويعيد السطر [5.5] الناتج الى ذلك الجزء من البرنامج الذي حدث فيه استدعاء هذه الدالة . وكنتيجة لذلك نرى بالسطر [4.11]، ان قيمة المتغير circleSurfaceArea قد اصبحت تلك القيمة المسندة اليه من ذلك الاستدعاء.

ان الدالة بالمثال [5] كاملة عدا عن امر واحد . فنحن لم نقم بتعيين "نوع – Type" البيانات العائد منها . وذلك مطلب اساسي حتى يتمكن المجّمع من القيام بعمله، وما علينا الا الطاعة والتنويه عن النوع على ان يكون عددي كسري[6.1] .

```
[6]
float circleArea(float theRadius)
{
  float theArea;
  theArea = 3.1416 * theRadius * theRadius;
  return theArea;
}
```

كما هو ملاحظ ان اول كلمة بالبيان [6.1] تصرح عن نوع البيانات العائد من هذه الدالة –قيمة المتغير theArea – وهي هنا من صنف عدد كسري. و من واجبك كمبرمج حريص، الحرص على التصريح عن متغير circleSurfaceArea المدرج داخل نطاق

main() ان يستقبل نفس نوع البيانات المتفق اي عدد كسري، وذلك لمنع المجّمع من اصدار تلك الرسائل المزعجة. ولا يشترط دوماَ ان تكون هناك معاملات للدالات. وحتى عندئذ يشترط ان تكون هناك اقواس ولو كامن فارغة.

```
[7]
int throwDice()
{
  int noOfEyes;

  // Code to generate a random value from 1
to 6

  return noOfEyes;
}
```

لايشترط ان تعيد الدالات اي قيمة **return a value**. فمتى ما كانت الدالة كذلك اصبح تصنيفها من نوع **void** . عندها يكون بيان العائد **"return"** اختياري. متى ما استخدمت مثل تلك الدالات احرص على ان تكون خانة القيمة العائدة **void**.

```
[8]
void beepXTimes(int x);
{
  // Code to beep x times.
  return;
}
```

عندما يكون للدالة المستخدمة اكثر من معامل مطلوب Arguments كالدالة **pictureSurfaceArea()** وهي تستخدم معاملان كمال بالمثال**[9]** ، عندها يجب ان نستخدم فاصل بين العوامل المتعددة وذلك بوضع فاصلة اعتيادية"," ببين العوامل.

```
[9]
float pictureSurfaceArea(float theWidth, float
theHeight)
{
  // Code here
}
```

ان الدالة الرئيسة **main()** تتطلب ارجاع قيمة مرتجعة **return value**، وهي ترجع قيمة صحيحة **integer**، ويجدر بها ارجاع قيمة صفر (**zero, [10.9]**) للتنوية عن ان الدالة انجزت عملها دونما اخطاء.

وبما ان الدالة **main()** ترجع قيمة صحيحة، يجب عندها كتابة نوع القيمة المرتجعة **int** قبل كتابة اسم الدالة كما يلي في **[10.1]** . والآن لنسرد كافة البنى البرمجية بالصندوق من الصفحة التالية.

```
[10]
int main()
{
  float pictureWidth, pictureHeight,
pictureSurfaceArea,
        circleRadius, circleSurfaceArea;
  pictureWidth = 8;
  pictureHeight = 4.5;
  circleRadius = 5.0;
  pictureSurfaceArea =  pictureWidth *
pictureHeight;
  circleSurfaceArea = circleArea(circleRadi
us);      // [10.9]
  return 0;
}

float circleArea(float theRadius)
// [10.13]
{
  float theArea;
  theArea = 3.1416 * theRadius * theRadius;
  return theArea;
}
```

كما ترى في**[10]** ، لدينا دالة **main() [10.1]** ودالة اخرى معرفة من قبلنا**[10.13]** . سيظل المجّمع في حيرة من امره عند قيامنا بتجميع وتجّميع البرنامج . ففي السطر **[10.9]** سيدعي عدم معرفته باي دالة اسمها **circleArea()** .يا ترى لماذا؟

من الواضح ان المجّمع قد بداء بتفسير الدالة الرئيسة **main()** ثم تواجه فجأة مع ما لا يعرف كيف يتعامل معه. عندها توقف عن العمل المناط به وأظهر لك رسالة تحذيرية.

حتى ترضي المجّمع ليتجاوز هذه المشكلة، قم بوضع التصريح عن الدالة في موقع اعلى من بيان الدالة الرئيسة **[11.1] int main()** . لاشيء صعب حيال ذلك، عدا انه مثل ما ورد بـ **[10.13]** فهو هنا إيعاز منتهي بفاصلة منقوطة.

```
[11]
float circleArea(float theRadius);  //
function declaration

int main()
{
```

ملحوظة : بقية البرنامج كما هو في المثال**[10]**. وقريبا سنركم ونشغّل هذا البرنامج، حالياَ لدينا اشياء من هنا وهناك.

من هنا وهناك

فأثناء كتابة البرامج، ينصح دوماَ باعتماد اعادة استخدام الشفرة البرمجية **code reuse**. ففي برنامجنا هذا يمكننا اضافة دالة احتساب مساحة المستطيل **rectangleArea()**، كما هو موضح بالمثال**[12]**، ويمكن استدعائها من نطاق الدالة الرئيسة **main()**. هذا الامر مفيد حتى لو كان التعامل مع هذه الدالة سيتم لمرة واحدة فقط وقت تشغيل البرنامج.

ستجد ان الدالة الرئيسة **main()** اصبحت قرائتها ايسر ، ومتى ما قمت بتقصي الاخطاء سيكون ايجادها اسهل، وقد تضع يدك على خطاء برمجي داخل احد تلك الدوال، بدلاً من ان تطوف بكافة الكتل المكونة للبرنامج، كل المطلوب منك هو تفحص جزئية بسيطة في الدالة، والشكر موصول لتلك الاقواس المعقوفة.

```
[12]
float rectangleArea(float length, float width)
{
  return (length * width);
}
```

كما ترى في حالات بسيطة مثل هذه الحالة من الممكن ان تجد إيعاز واحد تقوم عليه بنية الدالة، وذلك لاجراء عمليتي الحساب والارسال دفعة واحدة. لقد استخدمنا متغير **theArea** دون حاجة فعلية له إلا لاستعراض كيفيه التصريح عن متغير داخل نطاق الدالة.

فبالرغم من سهولة الدالات التي عرفّناها بانفسنا في هذا الفصل، الا انه من المهم التنبه لامكانية تغيير الدوال دون احداث صدمة في أداء البرنامج طالما لم تغير السطور الاولى من تصاريح القيم العائدة وانواع معاملات الدوال.

كمثال يمكنك تغيير اسماء المتغيرات داخل النطاق المحلي للدالة **variable scope**، ومع ذلك ستظل الدالة تؤدي عملها المطلوب –ولن يكون لذلك اي اثر اعتراضي باجزاء البرنامج–.

ويمكن لمبرمج آخر كتابة بنية تلك الدالة، حيث يمكنك استخدامها دون التعمق في فهم كيفيه عملها الداخلي، فكل ما هو مطلوب منك هو ان تتعرف على كيفيه استخدامها. وذلك يعني معرفة:

– اسم الدالة
– قيمة وترتيب المعاملات والمعطيات وانواعها المستخدمة
– نوع القيمة العائدة/المسترجعة من تلك الدالة (قيمة ناتج احتساب مساحة سطح مستطيل مثلا)، ونوع الناتج عددي صحيح أو كسري

في مثالنا السابق تردك الاجابات تباعاَ كما يلي:

– اسم الدالة **rectangleArea**

– لها معاملان، كلاهما عدد كسري، المعامل الاول يمثل الطول، والمعامل الثاني يمثل العرض.

– الدالة تعود بقيمة مسترجعة، ونوعها عدد كسري (ويمكن معرفة ذلك من اول كلمة في التصريح [12.1])

ان الشفرة البرمجية المكونة للدالة محجوبة عن كافة اجزاء البرنامج والدالات البرمجية الاخرى، ويعد ذلك احد اهم مميزات لغة Objective-C . ففي الفصل الخامس سنناقش طبيعة هذا السلوك من الحجب. حالياً، علينا تشغيل Xcode وتجّميع البرنامج السابق [11] .

الطباعة على الشاشة

لقد تقدمنا بشكل جيد مع برنامجنا، ولكننا الى الان لم نناقش كيفية عرض نتائج العمليات التي أجريناها. هناك عدة خيارات لعرض النتائج على الشاشة.

في هذا الكتاب، سنستخدم الدالة التي توفرها Cocoa وهي دالة NSLog(). انها دالة فعالة وبسيطة، فلست بحاجة لكتابة اوامر منخفضة المستوى لتتمكن من عرض نتائجك على الشاشة. هذه الدالة تقوم بذلك.

ان دالة NSLog() مصممة في الاصل لعرض رسائل الاخطاء، وليس لعرض نتائج البرنامج. على اية حال استخدامها من السهولة بحيث ادرجناها في كتابنا لعرض النتائج. بمجرد ان تصقل خبراتك مع Cocoa ستتدرج لاستخدام تقنيات اكثر ملاءمة. والان لنتفحص دالة NSLog() المستخدمة.

```
[1]
int main()
{
  NSLog(@"Julia is a pretty actress.");
  return 0;
}
```

سيقوم الايعاز في المثال [1] باخراج النص التالي .

```
"Julia is a pretty actress."
```

ان كافة النصوص مابين علامتي التنصيص " ... " ذات البادئة @ يطلق عليها اسم (سلسلة الحروف string) . وبالاضافة الى النص ذاته، يمكن لدالة NSLog() ان تطبع عدة معلومات اضافية، كالتاريخ الحالي واسم البرنامج القائم. كمثال، مخرجات هذه الدالة الكاملة على نظامي ستكون :

```
2005-12-22 17:39:23.084 test[399] Julia is
a pretty actress.
```

يمكن لسلسلة الحروف النصية هذه ان تكون بقيمة صفر أو اكثر.

ملحوظة: الامثلة التالية تعرض الدوال الاكثر اهمية دون عرض للدالة الرئيسة main() كما يلي .

```
[2]
NSLog(@"");
NSLog(@" ");
```

ان الايعاز بالسطر [2.1] يحتوي سلسلة حروف محتواها يساوي قيمة صفر وهي تدعى سلسلة حروف فارغة empty string (اي ان طولها يساوي صفر) . الايعاز بالسطر [2.2] لا يرى ان سلسلة الحروف فارغة –بالرغم من ان شكلها كذلك– بل يراها سلسلة حروف تحتوي فراغ واحد single space، لذا قيمتها تساوي ١ .

هذه السلاسل تستخدم بضع حروف من الرموز المتتالية وهي تستخدم لمهام معينة داخل سلاسل الحروف. كمثال على ذلك استخدام الحرف \n لاجبار السلسلة أن تعرض سلسلة حروفها بسطر جديد . هذا الترميز اختصار لكلمة new line character .

```
[3]
NSLog(@"Julia is a pretty \nactress.");
```

الان سيكون الخرج كما يلي –مع التركيز على ما يهمنا فقط–

```
Julia is a pretty
actress.
```

ان الخط المائل "\" بالمثال[3.1] يدعى ترميز هروب escape character وهو يستخدم لاعلام الدالة NSLog() بأن الحرف التالي في السلسلة غير قابل للعرض، إن الحرف الخاص الغير معروض في هذا المثال هو "n" وهو يعني للدالة عرض مايلي من حروف السلسلة بسطر جديد .

> في حالات نادرة قد ترغب بطباعة هذا الخط المائل "\" بالذات على الشاشة، فما هو العمل؟

يمكنك تكرار وضع الخط المائل خلف (او امام) ذلك الخط المائل

–اي تكرره مرتان–. هذا الايعاز يخبر الدالة **NSLog()** ان الحرف التالي من الخط المائل مطلوب للطباعة على الشاشة وان اي حرف خاص آخر سيكون قيد الاهمال. هنا مثال:

```
[4]
NSLog(@"Julia is a pretty actress.\\n");
```

البيان السابق يعرض ما يلي على الشاشة.

```
Julia is a pretty actress.\n
```

حتى الان قمنا باستعرض السلاسل الثابتة، دعنا نطبع القيم الناتجة من حسابتنا الى الشاشة.

```
[5]
int x, integerToDisplay;
x = 1;
integerToDisplay = 5 + x;
NSLog(@"The value of the integer is %d.",
integerToDisplay);
```

لاحظ وجود كل من : سلسلة حروف، فاصلة واسم متغير مابين الاقواس. لقد احتوت سلسلة الحروف على رمز عجيب **%d**: انه رمز خاص كالخط المائل، فعلامة النسبة المئوية وحرف **d** تمثل حرف من الحروف الخاصة للدالة. عند إرفاق الرمز المئوي بحرف **d** (وهو اختصاراً لنوع **decimal number** العددي)، اثناء تنفيذ البرنامج سيتم استبدال ذلك الحرف بالقيمة التي يدل عليها ذلك المتغير **integerToDisplay** ، ان تشغيل البرنامج يعطي النتيجة التالية.

```
The value of the integer is 6.
```

ولعرض عدد من نوع كسري، يجب استخدام الحرف الخاص **%f** بدلاً من **%d**.

```
[6]
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@"The value of the float is %f.",
floatToDisplay);
```

لتحديد دقة عرض العدد الكسري يعود لك الامر في تحديد عدد الخانات المتاحة بعد الفاصلة . فلعرض خانتين بعد الفاصلة ادرج رقم ٢ مابين العلامة المئوية ورمز الكسر **f**.

```
[7]
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@"The value of the float is %.2f.",
floatToDisplay);
```

قد ترغب بانشاء جدول للقيم مع الحسابات المتكررة. تخيل جدول للتحويل مابين معطيات درجات الحرارة بالدرجة المئوية وما يقابلها بالفهرنهايت.

اذا اردت عرض القيم بشكل صحيح ومرتب عليك عرض البيانات ضمن عمود عرضه ثابت. ويمكنك تحديد العرض من خلال استخدام القيم من نوع عدد صحيح **integer** وذلك بادراج العدد مابين العلامة المئوية ٪ وحرف الكسر **f** (او العلامة المئوية ٪ وحرف **d**) على كل في حال كان العرض المسند اصغر من العرض المتاح للرقم الكسري، عندها تكون الاولوية للعدد المسند.

```
[8]
int x = 123456;
NSLog(@"%2d", x);
NSLog(@"%4d", x);
NSLog(@"%6d", x);
NSLog(@"%8d", x);
```

المثال[8] السابق يعطي الخرج التالي:

```
123456
123456
123456
  123456
```

ان الايعازان [8.1, 8.2] يسندنان مساحات قليلة من الخانات المتاحة للاعداد المعروضة، ولكن المساحة المتاحة مأخوذة على كل حال. فقط الايعاز [8.4] يسند مساحة اعرض من القيمة، لذا يمكننا رؤية عرض المسافات المتاحة للاعداد والتي تدل على حجز زائد عن مساحة احتواء العدد. يمكننا ايضاً تضمين مواصفات مسافة العرض مع الاعداد الكسرية.

```
[9]
float x=1234.5678
NSLog(@"Reserve a space of 10, and show 2
significant digits.";
NSLog(@"%10.2d", x);
```

وبالتأكيد يمكن عرض اكثر من قيمة واحدة، أو خليط من القيم المتنوعة. و يجب عليك عندها تحديد الصنف الصحيح للقيم – سواء كانت صحيحة **int** أو كسرية **float**– باستخدام الحروف الخاصة d% وf%.

```
[10]
int x = 8;
float pi = 3.1416;
NSLog(@"The integer value is %d, whereas
the float value is %f.", x, pi);
```

من الضروري استخدام الرمز الصحيح مع نوع صنف المتغير، في حال قيامك بإستغفال المجّمع للقيمة الاولى فإنك لن تنجو محاولتك حيث سنعرض القيمة التالية لعدم العرض. كمثال،

```
[10b]
int x = 8;
float pi = 3.1416;
NSLog(@”The integer value is %f, whereas
the float value is %f.”, x, pi);
```

خرج هذه الشفرة الناتج كالتالي:

```
The integer value is 0.000000, whereas the
float value is 0.000000.
```

لدينا سؤال واحد واجابة واحدة قبل المضي بتشغيل برنامجنا الاول. وهو كيف إستطاع برنامجنا التعرف على دالة **NSLog()** ؟

في الحقيقة برنامجنا لن يستطيع التعرف عليها الا اذا عرفناه بها اولا. وللقيام بذلك، علينا اخباره بأن يأمر المجّمع بجلب مكتبة من الدوال المفيدة، والتي من ضمنها دالة **NSLog()** من خلال الايعاز التالي:

```
#import <Foundation/Foundation.h>
```

هذا الايعاز يجب ان يكون اول أمر مكتوب بالشفرة البرمجية للبرنامج. والان اذا وضعنا كل ما تعلمناه خلال هذ الفصل، سيكون لدينا الشفرة البرمجية التالية، والتي سوف نقوم بتنفيذها بالفصل التالي.

```
[11]
#import <Foundation/Foundation.h>
float circleArea(float theRadius);
float rectangleArea(float width, float
height);

int main()
{
  float pictureWidth, pictureHeight,
pictureSurfaceArea,
        circleRadius, circleSurfaceArea;
  pictureWidth  = 8.0;
  pictureHeight = 4.5;
  circleRadius  = 5.0;
  pictureSurfaceArea = rectangleArea(pictur
eWidth, pictureHeight);
  circleSurfaceArea  = circleArea(circleRad
ius);
  NSLog(@”Area of circle: %10.2f.”,
circleSurfaceArea);
  NSLog(@”Area of picture: %f. “,
pictureSurfaceArea);
  return 0;
}
// continued ...
```

```
float circleArea(float theRadius)
// first custom function
{
  float theArea;
  theArea = 3.1416 * theRadius * theRadius;
  return theArea;
}

float rectangleArea(float width, float height)
// second custom function
{
  return width*height;
}
```

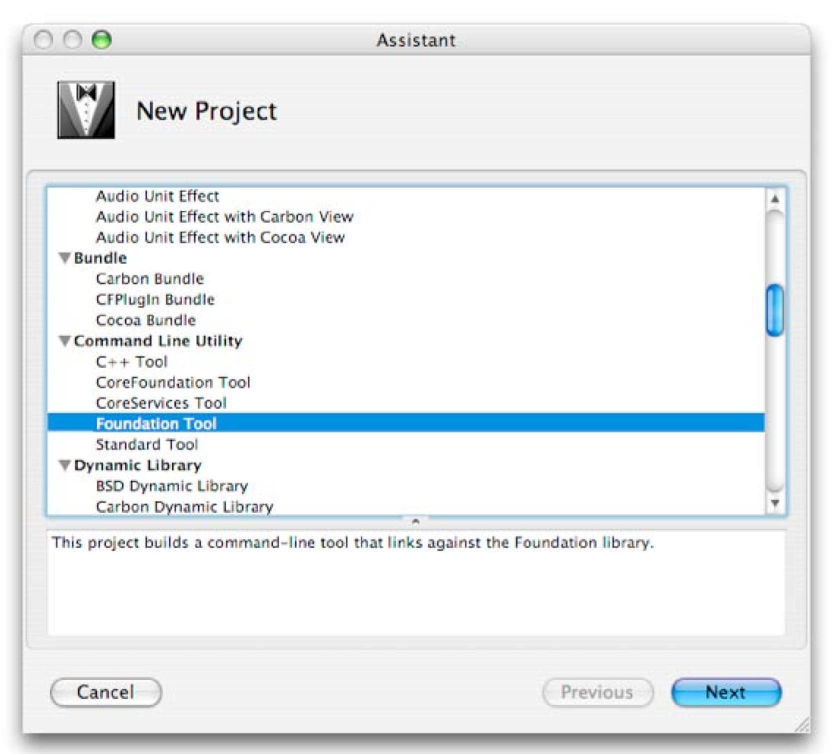
تجميع وتشغيل البرنامج

حتى الان تعد كافة الشفرات البرمجية التي استعرضناها عبارة عن نصوص قابلة للقراءة من قبلنا نحن البشر، ولو انها ليست مصُاغة بالشكل البديهي المعتاد، سيعاني الماكنتوش الامرين لو حاول التعامل معها كما هي. في الواقع لن يمكنه تحريك ساكن على الاطلاق!

يوجد برنامج خاص يدعي المجّمع **compiler** وهو ضروري لتحويل وتفسير تلك النصوص الى شفرة واكواد تشغيلية قابلة للتنفيذ من قبل الماكنتوش. يعد المجّمع جزء من بيئة البرمجة التي تزودك بها أپل مع **Xcode**. ويفترض انك قد قمت بتثبيت **Xcode** الذي جاء مع نظام التشغيل. على كل حال احرص على تحديثه باخر اصدارت أپل على العنوان التالي **http://developer.apple.com** (والتسجيل اجباري).

الان شغل برنامج .Xcode الذي ستجده داخل مجلد التطبيقات بمجلد المطورين. واذا كانت هذه المرة هي الاولى لتشغيل البرنامج عندها سيطرح عليك بضعة اسئلة. قم بقبول الإعدادات الافتراضية، حتى تتمكن من البدء ويمكنك دوماَ تغيير تلك الاعدادات وتخصيصها حسب احتياجاتك.

لتبدء فعلياَ، اختر مشروع جديد New Project من قائمة ملف File menu والتي توجد بأعلى الشاشة. سيظهر لك صندوق حوار سارداَ لائحة وافية بخيارات المشاريع التي يمكنك الانطلاق منها لتطوير مشروعك.



ان المساعد المضّمن بالبرنامج يعينك على انشاء مشاريعك بشكل اسرع

نرغب حالياَ بانشاء برنامج بسيط جداَ مستخدمين Objective-C ، (دون التطرق لاستخدام واجهة البرمجة الرسومية GUI) لذا توجة ادنى القائمة واختر "اداة اساسية Foundation Tool" حيث تجدها داخل مصنف ادوات سطر الاوامر Command Line Utility.

اطلق اسماَ لمشروعك

اطلق اسماَ للمشروع وحدد موقع حفظه داخل قرصك الصلب. واتمم اجراءات الانشاء بالضغط على ازرار الانتهاء. لقد اسمينا المشروع justatry ولك حرية التسمية كيفما شئت. ان مشروعنا قيد الانشاء معد للعمل من خلال وحدة طرفية Terminal تستخدم

سطر الاوامر كاسلوب تفاعلي مع الحاسب. وحتى تتجنب بعض العوائق قم باختيار اسم من كلمة واحدة لمشروعك.

> ان من المتعارف عليه تسمية ملفات الأدوات التي تشّغل من خلال الوحدات الطرفية باسماء يكون حرفها الاول صغير. بينما البرامج التي تشّغل من خلال الواجهة الرسومية باسماء حرفها الاول كبير.

تواجهك نافذة حالياَ، ستراها كثيراَ ما دمت بحقل البرمجة. هذه النافذة مكونة من اطارين (قطاعين).

يمثل الاطار الايسر فيها عرض لما لديك من ملفات ومجموعات وذلك لتمكينك من النفاذ الى كافة الملفات المساهمة في تكوين برنامجك. حالياَ لا يوجد ملفات كثيرة.

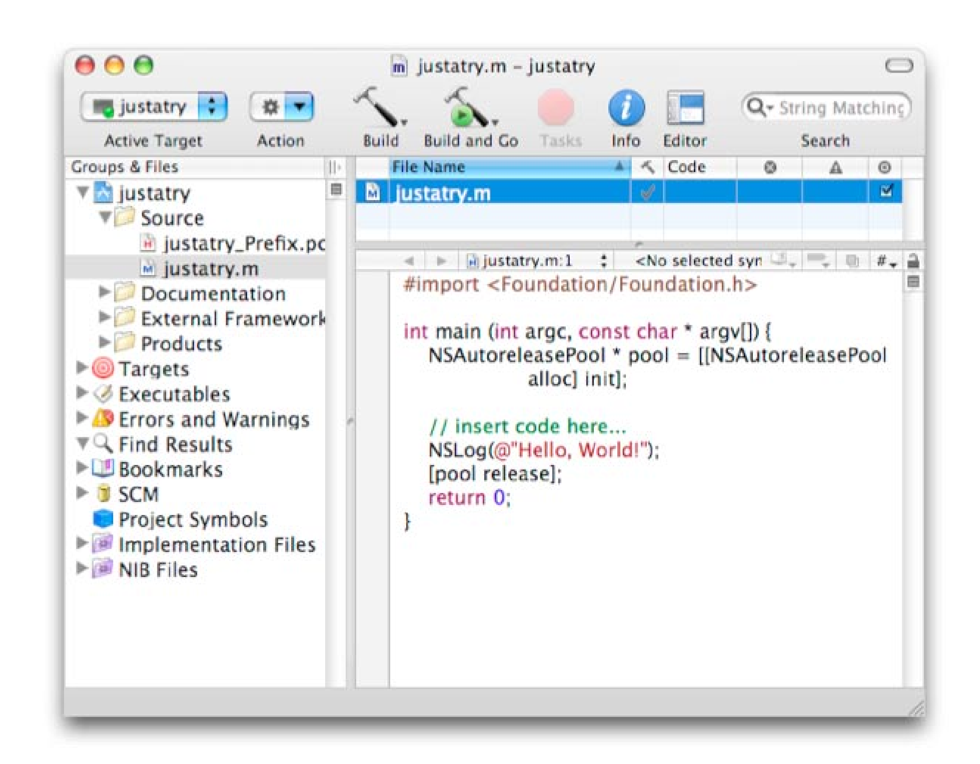
> في المستقبل ومع تعاملك مع الواجهة الرسومية ستجد العديد من الملفات هنا.

يُعد الاطار الايسر المنظم الذي يعرض لك مصادر الملفات حيث يعينك على تصنيفها وتجميعها. سواء استخدمت الواجهة الرسومية أو اللغات الاخرى المتاحة قيد التعامل.

هذه الملفات تجّمع وتحفظ داخل الاطار بمجلدات افتراضية، فما ان تقوم بعملية تصفح من خلال برنامج (**Finder**) الذي يقوم بتصفح وادارة الملفات على القرص الصلب لجهاز الماكنتوش، ستجد ان تلك الملفات جميعها محفوظة على مستوى جذري واحد **Flat Level** داخل مجلد مشروعك. **Xcode** يوفر لك ميزة الترتيب الافتراضي من خلال فكرة المجموعات **"groups"** والهدف منها تنظيمي بحت.

قم بالضغط على مجموعة المصادر المسماة **"Source"** والتي ستجدها بالاطار الايسر حيث توجد الملفات والمجلدات . داخل هذا المجلد ستجد ملف –مسمى باسم مشروعك– وهو هنا **justry.m** وهل تتذكر كيف ان كل برنامج يتطلب وجود دالة رئيسة بإِسم **main()** ؟ في الحقيقة ذلك هو الملف المحتوي للدالة الرئيسة.

سنقوم لاحقاَ خلال هذا الفصل باجراء تعديلات على الملف حيث سيشمل الشفرة البرمجية التي يتكون منها برنامجنا. اذا قمت بفتح الملف المسمى **justatry.m** وذلك بالضغط على ايقونة الملف عندها ستجد مفاجأة سعيدة بانتظارك. فقد قامت أپل بكتابة الدالة الرئيسة **main()** بالنيابة عنك.



Xcode يستعرض دالة main() الرئيسة

تفحص الشفرة البرمجية التي كتبت بالنيابة عنك، وهي موضحة بالمثال [1] وحاول التعرف على ما تألفه من اوامر ودالات رأيناها سابقاَ :

```
[1]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[]) //
[1.3]
{
  NSAutoreleasePool * pool =
[[NSAutoreleasePool alloc] init]; // [1.5]
  // insert code here...
  NSLog(@"Hello, World!");
  [pool release]; //[1.9]
  return 0;
}
```

سوف ترى :

– ايعاز جلب الدوال import الاجباري وامامه علامة المربع# هذا الامر الذي سيجلب لك دالة NSLog().

– دالة main() الرئيسة.

– الاقواس المعقوفة {} التي ستحتوي البنية البرمجية للدالة.

– تعليقات، تحثنا لكتابة شفرتنا البرمجية مكانها.

– ايعاز يستخدم دالة NSlog لطباعة سلسلة الحروف.

– ايعاز الأرجاع return ، الذي يعلن حين إنتهاء البرنامج .

توجد على اية حال عدة ايعازات غير مألوفة لديك:

– معاملات غريبة الشكل int argc, const char * argv[]، محصورة بين اقواس الدالة الرئيسة main() [1.3] .

– ايعاز بدء عمل لدالة غريبة اسمها NSAutoreleasePool بالسطر [1.5] .

– ايعاز آخر يحتوي كلمتي pool و release بالسطر [1.9] .

شخصياَ اشعر بعدم الرضا عندما يقوم مؤلفي الكتب بقذفي بمجموعة كبيرة من الرموز الغير مألوفة مع وعد قاطع بانها ستكون مفهومة ومألوفة بعد فترة. لذلك كما ترى ارتأينا تعريفك بمفهوم الدوال وجعلناه موضوعاَ مألوفاَ قبل الخوض بغمار التعريفات الاخرى التي قد تشكل صعوبة في الفهم.

لقد آلفت مفهوم الدوال وما هو دورها في تنظيم تركيب اجزاء البرنامج، وكيف ان اي برنامج يحتوي دالة رئيسة هي main() ، وكيف هو شكل هذه الدوال. ولكن عليّ الاعتراف بانه لا يمكنني شرح كل تراه بالمثال رقم [1] حالياَ على الاقل.

لذا آمل منك المعذرة حيث ساطلب منك تجاهل الايعازات الغير مألوفة لديك (اعني بذلك كل من [1.3, 1.5 , 1.9]) في الوقت الحالي. فهناك امور ومفاهيم عديدة تتطلب منك ان تألفها اولاَ بـ Objective-C مما يمكّنك من كتابة البرامج بسهولة اكثر .

المكاسب الجيدة هنا، هي واقع انتهائك من قراءة فصلين من اكثر الفصول صعوبة بهذا الكتاب، وانك مقبل على ثلاثة فصول سهلة نسبياَ قبل ان نتواجه مرة اخرى مع امور تتطلب قدراَ من التركيز النسبي .

في حال عدم رغبتك بمغادرة هذا الجزء الا بوجود تفسير حول ما سبق، ليكن لديك هذا التفسير المقتضب :

إن معاملات الدالة الرئيسة main مطلوبة متى تم تشغيل البرنامج من خلال الوحدة الطرفية Terminal. وبرنامجك كاي برنامج اخر يتطلب حجز قدراَ من الذاكرة المتاحة، ذلك القدر الذي سترغب البرامج الاخرى بحجزها لنفسها فور انتهاء برنامجك منها .

ومن واجبك كمبرمج، القيام بحجز ذلك القدر من الذاكرة التي يحتاجها برنامجك، ويترتب عليك ايضاَ مهمة ارجاعها للنظام فور انتهائك منها حتى يتيحها لمن يرغب باستخدامها . ذلك ما يقوم به الايعازان اللذان يحتويان كلمة "pool" داخلهما .

دعنا الان نشّغل البرنامج المقدم الينا من قبل أپل، في المثال [1]. اضغط الازرار الثاني الذي يحتوي صورة مطرقة ويظهر اسم Build بشزيط الادوات، وتوجه الى بند compile build وشغّل البرنامج.



ازرار Build and Go

سيتم تجّميع البرنامج ثم تشغيله، وسيتم عرض النتائج داخل اطار نافذة التوثيق Run Log window ، بالاضافة الى بعض المعلومات الاضافية. إن آخر جملة ستقرأها تفيد خروج البرنامج وتوقفه عن العمل بقيمة مقدارها صفر.

تلك القيمة التي انتجتها الدالة الرئيسة main()، والتي تم ذكرها بالفصل الثالث [7.9] لذا من الواضح ان برنامجنا قد نفذ كافة المهام المدرجة لدية حتى آخر سطر، ولم يتوقف بشكل مبكر. حتى الان الامور تسير على ما يرام!

نعود الى مثالنا [1] لنرى ما سيحدث لو كان هناك مشكلة برمجية bug.

كمثال قمت باستبدال دالة NSLog بدالة اخرى ولكنني "اغفلت عمداً" اضافة الفاصلة المنقوطة التي تعلن انتهاء السطر.

```
[2]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool * pool =
[[NSAutoreleasePool alloc] init];
  // insert code here...
  NSLog(@"Julia is a pretty actress")   //
Whoops, forgot the semicolon!
  [pool release];  //[2.9]
  return 0;
}
```

لتجّميع وبناء هذا البرنامج، اضغط ايقونة البناء الموجودة بشريط الأدوات. ستظهر لك دائرة صغيرة حمراء امام الايعاز [2.9].

```
// insert code here...
NSLog(@"Julia is a pretty actress")
[pool release];
return 0;
}
```

Xcode يبليغك عن وجود خطاء

لقد قام Xcode بتبليغك عن وجود خطاء اثناء التجّميع، فتلك هي طريقته – وإذا ضغطت الدائرة الحمراء، سيظهر لك سطر من المعلومات المقتضبة اسفل شريط الأدوات شارحاَ سبب الاعتراض:

```
error: parse error before "release".
```

ان عملية التفسير مرحلة من احد المراحل التي يقوم بها المجّمع: فهو يسبر كافة سطور الشفرة البرمجية مقّيماَ ما اذا كانت مقبولة لديه أو مرفوضة.

والامر يعود لك لتزويد المجّمع ببعض الحلول، فنحن نقدم له يد العون لتقبل بعض الجزئيات، فإيعاز جلب المصادر **import** مصرح عنه باستخدام رمز المربع #، وللتنويه عن انتهاء سطر الشفرة البرمجي إستخدمنا الفاصلة المنقوطة [2.8] ، وبينما يسبر المجّمع الايعاز [2.9] سيلحظ امراَ يشكل خطاء يستدعي منه التوقف.

على كل حال لن يدرك المجّمع مكمن حدوث الخطاء بذلك السطر، بل سيقوم بالقاء الخطاء على ذلك السطر الذي يفتقد الفاصلة المنقوطة.

والعبرة هنا هي ان المجّمع يحاول اعطاءك معلومات حول الاخطاء التي يصادفها، الا انه في مواضع كثيرة يكون غير دقيق في وضع اليد عليها (هو ليس دقيق بالضرورة ولكنه قريب من تعيينها).

قم باصلاح الخطاء الوارد بالبرنامج باضافة الفاصلة المنقوطة التي اغلفناها عمداً وأعد تشغيل البرنامج، وكن على يقين من انه يعمل بالشكل المتوقع الذي قمت بترتيبة.

دعنا الان نأخذ الشفرة البرمجية التي تناولناها بالفصل السابق ولندرجها بشفرة هذا البرنامج الذي قدمته لنا أبل [1] ولنرى نتاجه بالمثال [3]

```
[3]


#import <Foundation/Foundation.h>

float circleArea(float theRadius);    //
[3.3]

int main (int argc, const char * argv[])
// [3.5]
{
  NSAutoreleasePool * pool =
[[NSAutoreleasePool alloc] init];
  int pictureWidth;
  float pictureHeight, pictureSurfaceArea,
        circleRadius, circleSurfaceArea;
  pictureWidth = 8;
  pictureHeight = 4.5;
  circleRadius = 5.0;
  pictureSurfaceArea =  pictureWidth *
pictureHeight;


// Continued ...
```

```
  circleSurfaceArea = circleArea(circleRadi
us);
  NSLog(@"Area of picture: %f.  Area of
circle: %10.2f.",
        pictureSurfaceArea,
circleSurfaceArea);
  [pool release];
  return 0;
}
float circleArea(float theRadius)
// [3.22]
{
  float theArea;
  theArea = 3.1416 * theRadius * theRadius;
  return theArea;
}

float rectangleArea(float width, float height)
// [3.29]
{
  return width*height;
}
```

خذ ما يتطلب من وقت لتفحص واستيعاب هيكلية هذا البرنامج. فلدينا هنا ترويسات **headers** وهي تقوم بالتصّريح عن دالات استحدثناها مثل **circleArea()** بـ **[3.22]** و **rectangleArea()** بالسطر **[3.29]** وهي التي تسبق ظهور الدالة الرئيسة **main()** بالسطر **[3.5]**، فتلك هي مواضعها الصحيحة.

لقد وضعنا البنية البرمجية للدالة الرئيسة **main()** حيث اخبرتنا أپل، عند تنفيذ تلك الشفرة البرمجية، سيكون الخرج كما يلي:

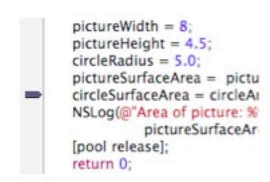
```
Area of picture: 36.000000. Area of circle:
78.54.

justatry has exited with status 0.
```

لقد قام البرنامج بتنفيذ التعليمات المطلوبة منه وانتهى من العمل منهياَ نفسه بان اعلن عن ذلك الحدث بقيمة صفر المعلنة. فعندما يبدأ البرنامج بالتوسع والتعقيد، عندها تصبح عمليات تقصي الاخطاء اكثر صعوبة.

لذا قد ترغب احياناَ بمعرفة ما يجري خلف الكواليس اثناء تشغيل البرنامج، ان **Xcode** يسهل عليك هذا الامر.

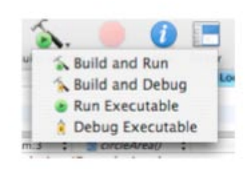
قم بالضغط على الهوامش الرمادية امام احد الايعازات التي ترغب بتفحص قيمها. سيقوم **Xcode** بادراج سهم رمادي امام ذلك الايعاز حيث يسجله كحدث يستوجب التوقف المؤقت **"breakpoint"** اثناء سير البرنامج ومروره بهذا السهم.



ادراج علامات التوقف المؤقت breakpoint داخل الشفرة البرمجية

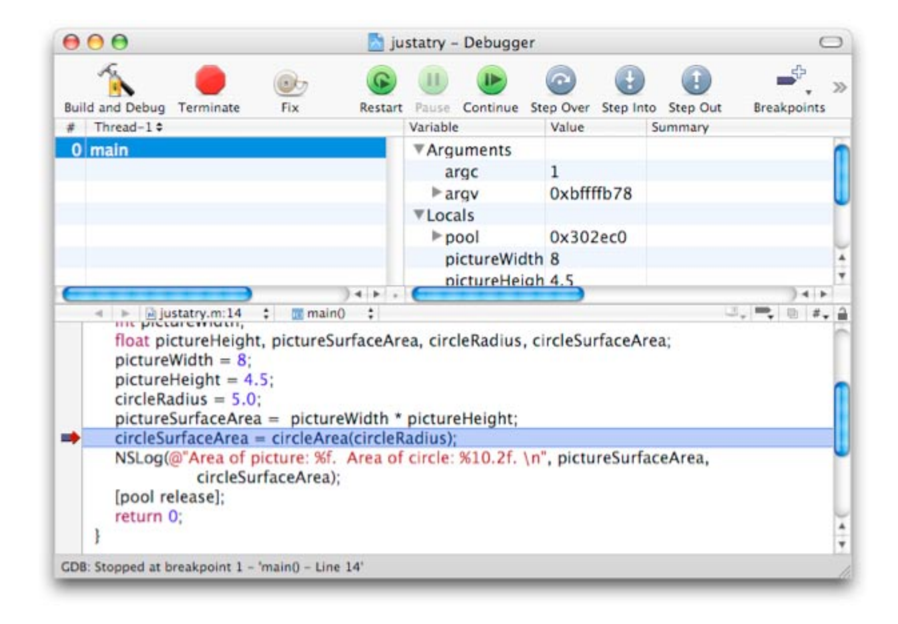
لاحظ انك ستتفحص قيم للمتغيرات بمرحلة تسبق ماقبل ذلك الايعاز الذي عينت عنده لحظة التوقف المؤقت، لذا غالباَ ما ستضع نقاط التوقف بعد ذلك الايعاز الذي ترغب بتفحص قيمته.

الان اضغط الفأرة ضغطة مستمرة داخل الأزرار الثاني بشريط الأدوات، عندها ستظهر لك قائمة منسدلة **pop-up menu**.



قائمة البناء والتشغيل Build and Go

اختر بناء وتقصي **Build and Debug**. وسترى النافذة التالية حيث يسمح لك **Xcode** بتنفيذ البرنامج خطوة خطوة مع عرض للمتغيرات وقيمها ايضاَ مع كل خطوة.



Xcode يتقصى الاخطاء البرمجية خطوة خطوة

يعمل البرنامج اثناء مرحلة التقصي هذه بشكل طبيعي ثم يتوقف عن تنفيذ الإجراءات حين يلتقي باحد علامات التوقف المؤقت. اذا قمت بالضغط على الاطار الاعلى يميناَ، ستتمكن عندها من رؤية القيم المخزنة لعدد من المتغيرات.

ان اي قيمة تم تغييرها منذ اخر علامة توقف مؤقت سيتم عرضها باللون الاحمر. وللسماح للبرنامج ان يكمل عمله استخدم ازرار الاستمرار. إن برنامج تقصي الاخطاء البرمجية **debugger** اداة ذات

اهمية قوية. فحاول ان تتعامل معه حتى تألف طريقة عمله.

لدينا الان كل ما نحتاجة لتصميم و كتابة البرامج، وتقصي أخطاء البرمجة وامكانية تشغيل البرامج البسيطة –منعدمة الواجهات الرسومية– على نظام Mac OS X .

في حال عدم رغبتك بتصميم برامج تستخدم الواجهة الرسومية، يجدر بك الآن ان تتزود بالمعارف والمعلومات المتاحة عن لغة البرمجة Objective-C حتى تتمكن من تطوير البرامج المتخصصة دون الحاجة لاستخدام الواجهة الرسومية للنظام. وفي الفصول التالية سوف نقوم بذلك التزود المعرفي حرفياً. ثم سنسبر إنشاء التطبيقات التي تستخدم الواجهة الرسومية للنظام. فاكمل القراءة!!

التعليمات الشرطية

في بعض الاحيان قد ترغب بجعل شفرتك البرمجية قادرة على توجية عملية سير اوامر البرنامج بناء على استيفاء شروط معينة. لذا توجد مفردات محجوزة لتحقيق هذا الامر.

```
[1]
int age = 42;
if (age > 30)    // The > symbol means
"greater than"
{
  NSLog(@"age is older than thirty.");
//[1.4]
}
NSLog(@"Finished."); //[1.6]
```

السطر [1.2] يظهر ايعاز استفسار الشرط **...if** والذي تنحصر مهمتة بتقّييم تحقيق شرط معين. سوف تتعرف على دور الاقواس المعقوفة والتي بدورها تحتوي الشفرة البرمجية التي يجب تنفيذها متى ما استوفيت صحة حقيقة الشرط المنطقي الصياغة و المحصور بين الاقواس.

هنا اذا استوفى الشرط الحقيقة **if the condition age > 30** عند ذلك سيتم طباعة سلسلة الحروف المعينة **[1.4]**. ثم سيتم طباعة

نص سلسلة الحروف المعينة بـ **[1.6]**، سواء تم استيفاء الشرط من عدمة ذلك يعود لان موقعها خارج نطاق الشرط، اي خارج الاقواس المعقوفة.

يمكننا ايضاَ تزويد استعلام الشرط بتعليمات اخرى في حال لم يتم الاستيفاء. باستخدام مفردة **if...else statement** بالمثال **[2]**.

```
[2]
int age = 42;
if (age > 30)
{
  NSLog(@"age is older than thirty.");
//[2.4]
}
else
{
  NSLog(@"age is not older thirty.");
//[2.7]
}
NSLog(@"Finished."); //[1.6]
```

سيتم طباعة النص **[2.7]** فقط في حال عدم تحقق الشرط، وذلك لن يتحقق هنا بالمثال **[2]**. وبالاضافة الى اقواس المفاضلة ما بين القيم الاكبر ">" أو الاصغر "<" الخاصة بالاعداد يوجد لديك معاملات اخرى متنوعة:

يساوي ==
اكبر من >
اصغر من <
اكبر من أو يساوي >=
اصغر من أو يساوي <=
لايساوي !=

تمعن بمعامل رمز مفاضلة المساواة هنا حيث لدينا علامتي يساوي متكررة "==" وهي من السهولة بحيث ان يتم ادراج واحدة فقط مع نسيان الاخرى عندها سيتم التعامل مع ذلك الايعاز كاسناد للقيمة بالرغم من اننا قد نرغب باجراء اختبار مفاضلة تساوي بين القيم. وذلك خطاء شبه شائع ومدعاة للارتباك وأحد اهم اسباب وجود اخطاء برمجية بالشفرات المصدرية للمبتدئين.

ردد معي بصوت عالي و مسموع:

لن انسى استخدام علامتي يساوي مثل هذه == متى رغبت باجراء مفاضلة تساوي بين القيم!

إن معاملات المفاضلة مفيدة جداَ متى رغبت باجراء سلسلة من الايعازات لعدد من المرات. وذلك سيكون موضوع حديثنا بالفصل التالي.

دعنا الان نناقش بعض المفاهيم التي قد يكون لها استخدام مثير. و لنتفحص بشكل اعمق عملية المفاضلة هذه. ان نتيجة المفاضلة تنحصر بين نتيجتين لا ثالث لهما: النتيجة إما تكون صحيحة أو غير–صحيحة / خاطئة **true or false**.

في Objective-C ، يتم تمثيل النتيجة الصحيحة أو غير–الصحيحة بإما **1** أو **0** على التوالي. حتى انه يوجد نوع خاص من التصنيفات اسمة BOOL تنحصر مهمتة باظهار هذه النتيجة. وحتى نقول ان قيمة النتيجة "صحيحة **true**" يمكنك اما كتابة **1** أو YES. ولاسناد قيمة غير–صحيحة / خاطئة يمكنك اما كتابة **0** أو NO

```
[3]
int x = 3;
BOOL y;
y = (x == 4);  // y will be 0.
```

من الممكن عمل اختبار حقيقة لأكثر من شرط. ويتم ذلك عندما يتطلب الامر استيفاء كافة الشروط، عندها نستخدم المعامل المنطقي AND ، وهو يرد دائما بعلامتي **ampersands: &&.** تتقدمه. اما عند اشتراط قيمة صحيحة متحققه لأحد الشروط، عندها نستخدم المعامل المنطقي OR ، وهو يرد دائما بعلامتي|| .

```
[4]
if ( (age >= 18) && (age < 65) )
{
  NSLog(@"Probably has to work for a 
living.");
}
```

من الممكن تعشيش عدة ايعازات اشتراطية **-nested condition-**. فهي ببساطة تتمثل بادراج شرط داخل الأقواس المعقوفة للاشتراط الاول. عندها سيتم تقييم الاشتراط الاول، ثم بعد استيفاء تحقيق حقيقة شروطة سيتم تقييم الايعاز الشرطي المعشش داخله وهكذا تباعاً.

```
[5]
if (age >= 18)
{
  if (age < 65)
  {
    NSLog(@"Probably has to work for a 
living.");
  }
}
```

التعليمات المتكررة

كافة الايعازات والاوامر التي مررنا بها في الشفرات البرمجية سابقاَ تنفذ مرة واحدة فقط. يمكننا دوماَ تكرار تنفيذ اوامر الدالات باستدعائها مراراَ وتكراراَ[1].

```
[1]
NSLog(@"Julia is a pretty actress.");
NSLog(@"Julia is a pretty actress.");
NSLog(@"Julia is a pretty actress.");
```

بناء على ذلك سيتم استدعاء وتنفيذ الدالة حسب عدد المرات التي وردت ولكن تكرار كتابة استدعاء الدالة داخل الشفرة يعد امراَ غير عملي. قفي بعض الاحيان يتطلب ان يتم تكرار تنفيذ الايعاز أو الدالة عدد معين من المرات يحدد وقت تشغيل البرنامج.

مثلها مثل لغات البرمجة الاخرى، تقدم لك لغة **Objective-C** عدة طرق لتحقيق ذلك فاذا حددت العدد المطلوب من المرات اللازمة لاجراء التكرار على الايعاز أو مجموعة الايعازات أو الدالات، عندها يمكنك تحديد ذلك الرقم مستخدماَ مفردة **for** الخاصة بإجراء حلقة تكرارية، وللعلم يجب ان يكون العدد صحيح **integer**، حيث لن تستطيع عمل خمس دورات ونص!

```
[2]
int x;
for (x = 1; x <= 10; x++)
{
  NSLog(@"Julia is a pretty actress.");
}
NSLog(@"The value of x is %d", x);
```

في المثال [2] ،سيتم طباعة سلسلة حروف محددة في [1.4] عدد ١٠ مرات. تم اولا انشاء متغير باسم x واسندت له قيمة مقدارها واحد. سيقوم الحاسب بتقييم القيمة ويفاضلها مع المعادلة x <= 10 التي تشترط ان تكون قيمة x اقل أو تساوي ١٠.

يستوفي هذا الاشتراط تحقيقة وتظل حلقة التكرار تحدث متى ما كانت قيمة اقل او تساوي ١٠ (x <= 10)، هذا الشرط متحقق حيث ان قيمة X تساوي ١ وهي اقل من ١٠ وسيظل امر التكرار قيد العمل.

اثناء اتمام كل دورة للحلقة سيتم زيادة قيمة المتغيرx بنسبة عددية تساوي ١ وذلك لوجود العداد ++x الذي يؤدي لزيادة قيمة x بكل دورة.

لاحقاَ ستصبح قيمة المتغيرx مساوية للعدد٢ وسيتم عمل تقييم للمفاضلة بين القيمة والمعادلة المحددة لشرط تكرارية الحلقة والتي ستظل تعمل حتى يتحقق الشرط وهو ان تكون قيمة x اقل أو تساوي ١٠. فما ان تصل قيمة المتغير x الى ١١ عندها يكون الشرط بالمعادلة قد تحقق، وبذلك يتم الخروج من حلقة التكرار.

اخر ايعاز يضمن لك ان تكون قيمة x مساوية لـ ١١ وليس ١٠ وهنا تنتهي الحلقة التكرارية.

في بعض الاحيان قد ترغب بمعدل زيادة اكبر في الخطوات من قيمة ١ حيث استخدمنا ++x لزيادتها. كل ما عليك عمله هو تغيير خطوات الزيادة "steps" بصيغة المعادلة في خانة الخطوات. المثال التالي[2] يقوم بتغيير القيم من درجات مئوية الى درجات فهرنهايت.

```
[3]
float celsius, tempInFahrenheit;
for (tempInFahrenheit = 0; tempInFahrenheit
<= 200; tempInFahrenheit =
tempInFahrenheit + 20)
{
  celsius = (tempInFahrenheit - 32.0) * 5.0
/ 9.0;
  NSLog(@"%10.2f -> %10.2f",
tempInFahrenheit, celsius);
}
```

وخرج البرنامج سيكون :

```
      0.00 ->    -17.78
     20.00 ->     -6.67
     40.00 ->      4.44
     60.00 ->     15.56
     80.00 ->     26.67
    100.00 ->     37.78
    120.00 ->     48.89
    140.00 ->     60.00
    160.00 ->     71.11
    180.00 ->     82.22
    200.00 ->     93.33
```

بالإضافة إلى **for..loop** تقدم لك Objective-C طرق اخرى لعمل حلقات التكرار من خلال المفردات :

```
while () { }
```

وأيضاَ

```
do {} while ()
```

إن الايعاز الاخير مشابة جداَ لحلقة **for-loop**. فهذا الأيعاز **do {} while ()** يبدأ عمله بتقييم القيمة قيد النظر. طالما كانت نتيجة التقييم غير–صحيحة تظل الحلقة تدور حتى يتم استيفاء الشرط.

```
[4]
int counter = 1;
while (counter <= 10)
{
  NSLog(@"Julia is a pretty actress.\n");
  counter = counter + 1;
}
NSLog(@"The value of counter is %d",
counter);
```

في هذه الحالة، ستكون قيمة المتغير **counter** تساوي ١١، فهل من اهمية لها بعد ذلك في البرنامج؟ –الامر عائد اليك..

ان استخدام ايعاز حلقة **do {} while ()** يعطي الفرصة لتنفيذ للشفرة البرمجية ما بين الاقواس المعقوفة مرة واحدة على الاقل.

```
[5]
int counter = 1;
do
{
 NSLog(@"Julia is a pretty actress.\n");
 counter = counter + 1;
}
while (counter <= 10);

NSLog(@"The value of counter is %d", counter);
```

في نهاية عمل الحلقة ستكون قيمة المتغير مساوية لـ ١١.

لقد اكتسبت مهارات برمجية متنوعة مؤخراَ، لذا سنقوم بمناقشة مواضيع اكثر صعوبة، ستتطلب قدراَ من التركيز والتطبيق. ففي الفصل القادم سنقوم بانشاء اول برنامج يستخدم واجهة التطبيقات الرسومية أو (GUI).

البرمجة بإِستخدام واجهة التطبيقات الرسومية

مع ازدياد خبرة التعامل بـ Objective-C سنكون مهيئين لمناقشة كيفية إنشاء برنامج مستخدمين فيه ما تقدمة واجهة التطبيقات الرسومية (GUI) من خدمات. وعليّ الاعتراف بامر مهم هنا وهو ان Objective-C لغة برمجية مستقاة من لغة C. فأغلب ما تم عرضه حتى الان يعد C الصرفه . لذا السؤال الذي يطرح نفسه هو مالفرق الذي تشكلة لغة Objective-C عن لغة C؟

يكمن الاختلاف في ذلك الجزء المتعلق بـ "Objective". فـ Objective-C تتعامل مع سمات ومفاهيم مجردة تعرف باسم الكائنات objects .فحتى هذه اللحظة كنا قد تعاملنا مع الاعداد. ولكن كما سترى تتعامل Objective-C في الاصل مع الاعداد على انها مفاهيم مجردة.

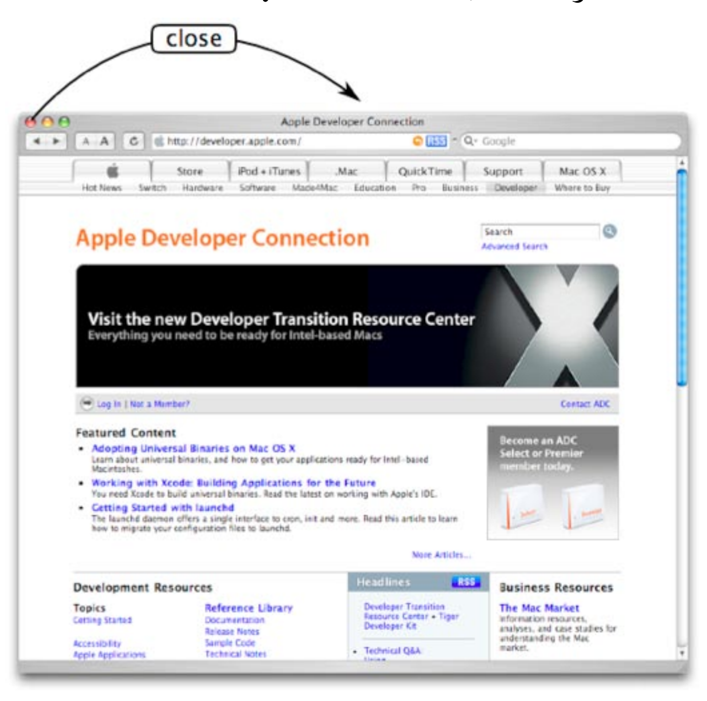
ذلك انها تمكنك من انشاء تلك الاعداد بالذاكرة وتمكنك من تطبيق المعاملات الحسابية عليها بتطبيق مختلف الدالات المناسبة بمعاملاتها. ذلك امر رائع عندما يتعامل برنامجك بالارقام (مثل

الالة الحاسبة) . لكن مالعمل عندما يتعامل برنامجك على سبيل المثال مع مقطوعات موسيقية و مكونات أخرى كلائحة باسماء المقطوعات، ولائحة باسماء المغنين؟ أو ان يدير برنامجك على سبيل المثال الحركة الجوية كنظام لادارة الطائرات والرحلات والمطارات؟ الن يكون من الرائع ان تتعامل مع تلك المعطيات من خلال Objective-C كما لو انك تتعامل مع الاعداد؟

هنا يأتي دور الكائنات . فمن خلال Objective-C يمكنك وضع توصيف لنوع الكائن البرمجي الذي ترغب بالتعامل معه، ومن ثم تكتب الشفرة البرمجية الملائمة للتعامل والتعاطي معه .

كمثال على ذلك دعنا نلقي نظرة على نافذة اي برنامج من إنشاء Objective-C ، كبرنامج Safari مثلاَ . تفحص نافذة البرنامج وانظر لتلك المساحة العلوية من النافذة حيث تجد تلك الازرار الثلاثة . ستجد ان الزر الاحمر منها يُغلق النافذة بمجرد ضغطك عليه بالفأرة . ترى مالذي يحدث عندما تغلق النافذة جراء ضغطك لذلك الزر الاحمر؟

لقد اغلقت النافذة ذاتها لانها تلقت رسالة message من ذلك الزر الاحمر تفيدها بوجوب اغلاق ذاتها . انها عملية رد فعل وتنفيذ لشفرة برمجية داخل النافذة كانت الرسالة هي المسبب لتنفيذ ذلك الجزء المتعلق بإغلاق النافذة لذاتها .بإختصار لقد تلقت النافذة رسالة تفيدها باغلاق ذاتها .



النافذة تتلقى رسالة إغلاق

فالنافذة ذاتها تعد كائن برمجي . حيث يمكنك سحبها هنا وهناك . ان تلك الازرار الثلاثة في الاصل هي كائنات برمجية يمكنك الضغط على اي منها . تلك الكائنات البرمجية مبنية

مع خاصية تمكنها من معرفتها لكيفية اظهار (رسم) ذاتها على شاشة المستخدم.

على كل جزء الإظهار هذا ليس سمة غالبة لكل الكائنات. فعلى سبيل المثال هناك كائنات تمثل عملية الاتصالات مابين برنامج سفاري وذلك الموقع الالكتروني ولكنها لاتمثل ذاتها بصرياً على الشاشة.

كائن نافذة يحتوى كائنات الازرار

ان اي كائن (النافذة مثلاً) يمكنه ان يحتوي على اي كائنات اخرى (الأزرار الثلاثة). يمكن لبرنامج سفاري ان يزودك باي عدد من النوافذ التي تريدها.

ترى هل كنت تتوقع ان:

١. قيام مبرمجي أپل ببرمجة كل تلك النوافذ مسبقاَ مستنفذين كافة مواردهم العقلية والحسابية حتى يلبوا طلباتك من النوافذ التي قد تحتاجها؟

٢. قاموا بوضع نموذج للنافذة ومكنوا برنامج سفاري من استخدام ذلك النموذج لانشاء نوافذ اخرى وقت الطلب؟

قطعاَ الاجابة الثانية هي الصحيحة.

لقد وضعوا بعض الشفرات البرمجية لتمثل نموذج للنافذة حيث حددوا التصرفات والسولكيات العامة الواجب اتباعها لتلك النافذة واطلقوا عليها تصنيف أو Class.

فعندما تنشيء نافذة جديدة فانك تعتمد في إنشائك لها على نسخة من ذلك التصنيف Class الذي يحتوي توصيف للنموذج الاصل للنافذة. فالتصنيف Class هنا يمثل مفهوم عن النافذة النموذج. مما يعني ان اي نافذة تراها امامك انما هي في الواقع مستنسخ instance من مفهوم النافذة النموذج. (في ذات السياق نستوعب ان العدد ٧٦ انما هو مستنسخ من مفهوم تصنيف الاعداد).

ان النافذة التي قمت بانشاءها تتخذ موقع ومساحة معينة من شاشة الماكنتوش. فإذا قمت بتصغيرها إلى المنصة Dock، ثم قمت بإستدعائها مرة اخرى سترى انها تعود الى ذات المكان الذي كانت متموضعة فيه. يا ترى كيف تعمل هذه الامور؟

سنرى ان التصنيف الاساسي يعّرف عدد من المتغيرات المناسبة والتي تتّحدّد مهمتها بحفظ موقع النافذة من الشاشة.

ان مستنسخ التصنيف class instance، أو الكائن الحالي يحتفظ بتلك القيم التي تخصه في متغيراته التي تخضع تحت تصرفه. لذا كل كائن نافذة يحتوي متغيرات محددة وبقيم خاصة بذلك

الكائن. ان عدة كائنات من النوافذ المختلفة يحتفظ كل كائن منها بقيم مختلفة لمتغيراتها الخاصة بها.

هذا التصنيف لم يقم بانشاء كائن النافذة فقط، لكنه مكنها من سلسلة من الانشطة والتصرفات (إجراءات **actions**) التي يمكن للنافذة ان تؤديها.

احد تلك الإجراءات، إجراء اغلاق النافذة لذاتها **close**. فمتى ما قمت بالضغط على ازرار الاغلاق الاحمر باحد النوافذ، سيقوم الازرار بارسال رسالة الى النافذة مفادها اغلقي ذاتك. ان الإجراءات التي يقوم بها الكائن تسمى **methods**. كما ترى انها شديدة الشبة بالدالات، وذلك لن يشكل مشكلة في التعلم والاستخدام اذا كنت متابع معنا حتى اللحظة.

عندما يقوم النموذج التصنيفي بإنشاء كائن نافذة من اجلك، فإنه في الواقع يحجز جزء من الذاكرة (RAM) كي يتم حفظ موقع النافذة من الشاشة، بالاضافة الى عدد من المعلومات الاخرى.

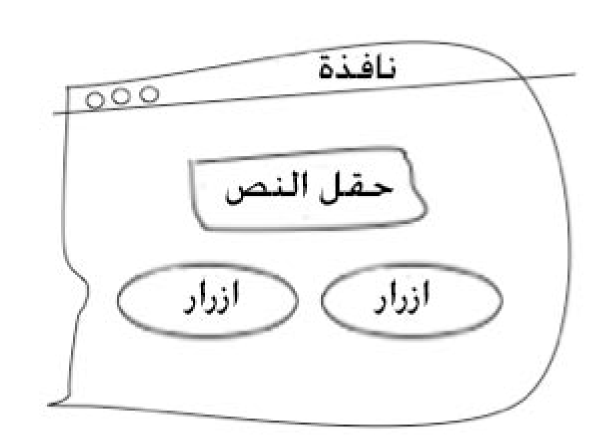
على كل حال عملية الاستنساخ هذه لا تقوم بنسخ كافة الشفرات البرمجية المكونة للنافذة كامر الاغلاق مثلا. هذا العمل سيبدد حيز كبير من الذاكرة المتاحة حيث سيتم نسخ كافة الشفرات البرمجية والإجراءات الخاصة بكل نافذة توجد هنا أو هناك.

الحل الامثل هو ان توجد الشفرة البرمجية المحددة لسلوكيات وتصرفات النافذة بموقع مركزي واحد فقط، حيث يتم مشاركة ومراجعة تلك الاوامر والسلوكيات مع اي كائن نافذة آخر ينتمي لذلك التصنيف.

كما أوردنا سابقاَ من امثلة، ستحتوي الشفرة البرمجية التي ستراها في هذا الفصل بعض السطور التي تهتم بحجز مواقع الذاكرة وتسريح الحجز حتى يتمكن النظام من استعادتها وتوظيفها مرة اخرى. وكما تم التنويه سابقاَ سنتطرق لشرح هذه الاسطر في مواضيع متقدمة لاحقاَ. لذا نرجو المعذرة...

في هذا سنقوم بإنشاء برنامج مكون من ازرارين اثنين وحقل نص **text field**. بحيث اذا قمت بالضغط على احد الازرار، سيتم عرض قيمة ما بـحقل النص. وعند الضغط على الازرار الاخر سيتم عرض قيمة اخرى بحقل النص. فكر في البرنامج وكأنه آله حاسبة مكونة من ازرارين ولايمكنها عمل اي عمليات حسابية...

بمجرد تعلم المزيد يمكنك صنع آلتك الحاسبة بالتأكيد، ولكننا نفضل التعلم خطوة خطوة.



هنا تصور عن البرنامج المزمع انشاءه.

في حال الضغط على احد تلك الازرار، سوف يقوم الازرار المضغوط بارسال رسالة. هذه الرسالة تحتوي اسم الاجراء **method** المزمع تنفيذه. ترى هذه الرسالة توجه الى من أو ماذا؟

في حالة النافذة، كانت رسالة الاغلاق موجهة الى كائن النافذة، والذي بدوره كان مستنسخ **instance** عن تصنيف النافذة اي **window class**. ما نحتاجة الان هو كائن لديه القدرة على استقبال الرسائل من اي من هؤلاء الازرارين، وكذلك القدرة لتوجيه حقل النص لعرض قيمة ما.

لذا في البدء علينا بإنشاء التصنيف الخاص بنا كنموذج، ثم نقوم بإستنساخه ككائن. هنا سيكون هذا الكائن هو المستقبل للرسائل التي سترسلها اليه تلك الازرار (انظر للرسم التوضيحي ادناه).

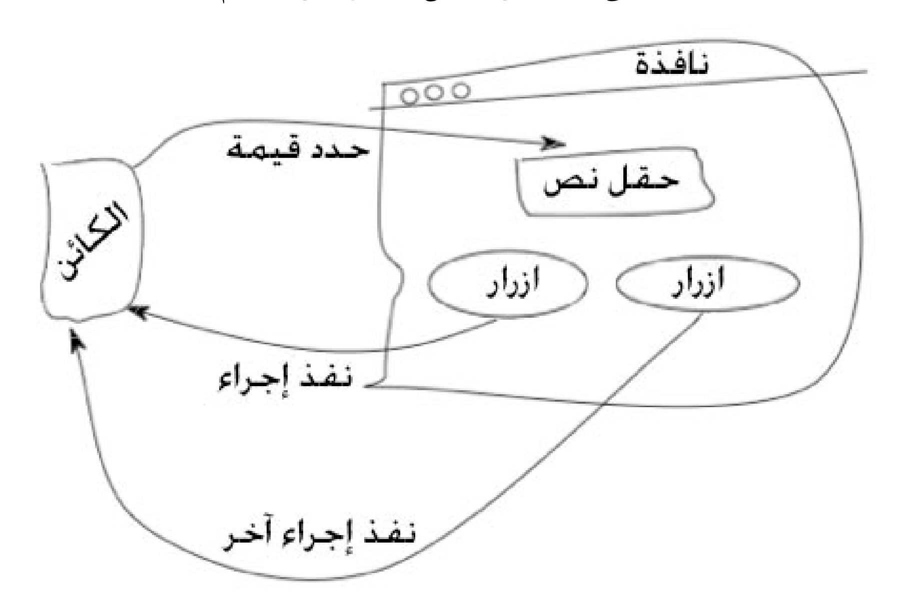
ان مستنسخنا عبارة عن كائن برمجي، مثله مثل كائن النافذة، ولكنه على النقيض من كائن النافذة فهذ المستنسخ غير ظاهر بصرياَ على الشاشة متى قمنا بتشغيل البرنامج. انه مجرد كائن برمجي موجود في ذاكرة الحاسب فقط.

ما ان يتلقى كائننا المستنسخ تلك الرسالة المرسلة اليه من احد تلك الازرار، سيتم عندها تنفيذ الاجراء المطلوب. والشفرة البرمجية لذلك الاجراء ستوجد بالتصنيف النموذجي الاصلي (اي انها ليست موجودة بالمستنسخ).

اثناء التنفيذ، سيقوم هذا الاجراء بارسال رسالة الى كائن حقل النص.بالاضافة الى اسم كائن حقل النص، تكون صيغة الرسالة المرسلة محتوية على ما يلي دوماَ: اسم الاجراء (الخاص بحقل النص).

فور تسلم حقل النص للرسالة يتم التنفيذ فوراَ. فنحن نريد من حقل النص ان يعرض قيمة نصية معينة، بناء على الازرار المضغوط. لذا ستحتوي الرسالة على مُعامل **argument** (قيمة)

بالاضافة الى اسم الكائن الذي سترسل اليه الرسالة، واسم الاجراء المطلوب تنفيذه حتى يتمكن حقل النص من القيام بدوره.



الرسم التوضيحي يوضح عملية تبادل الرسائل بين الكائنات داخل البرنامج.

هنا نعرض الطريقة العامة لارسال الرسائل بـ Objective-C دون وجود معاملات [1.1] وبوجودها[1.2]

```
[1]
[receiver message];
[receiver message:argument];
```

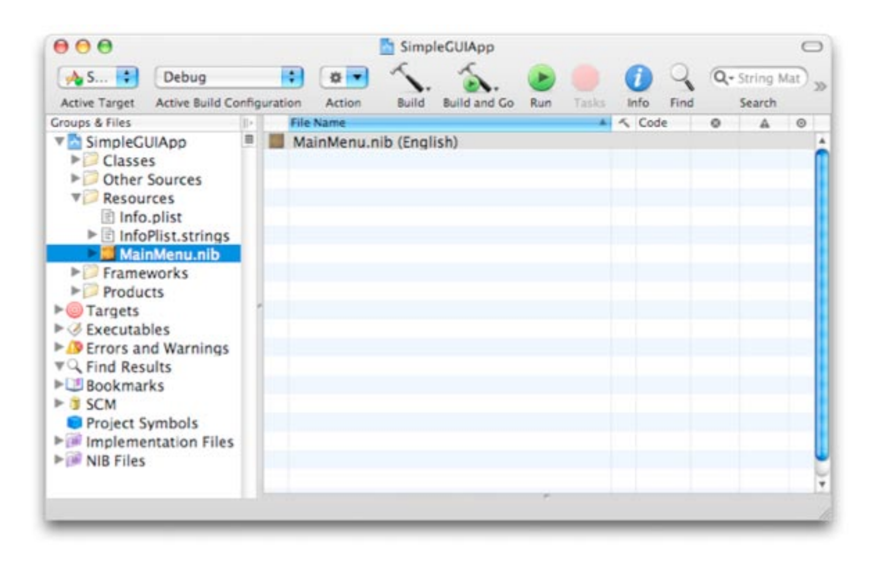
كما ترى من الايعازات، يكون القلب النابض للاجراء محصوراَ بين تلك الاقواس المربعة [...]; متبوعة بتلك الفاصلة المنقوطة الابدية.

داخل تلك الاقواس المربعة، يتم ذكر اسم الكائن المستقبل للرسالة اولاَ يتبعه اسم احد الاجراءات التي يمكنه تنفيذها. واذا ما تطلب الاجراء وجود اكثر من قيمة او معامل، وجب تزويده بها اجباراَ [1.2].

دعنا نتفحص كيفية عمل ذلك جدياَ.

شغّل Xcode وانشيء مشروع جديد. اختر نوع تطبيقات Cocoa Application المعششة داخل ترويسة التطبيقات Application heading.

اطلق اسم على المشروع (مع مراعاة قواعد التسمية التي تحثك على إستخدام حرف كبير Capital في بداية اسم التطبيق المستخدم للواجهة الرسومية). داخل اطار الملفات والمجموعات، افتح مجلد المصادر Resources واضغط متتالياَ على الايقونة المسماة MainMenu.nib.

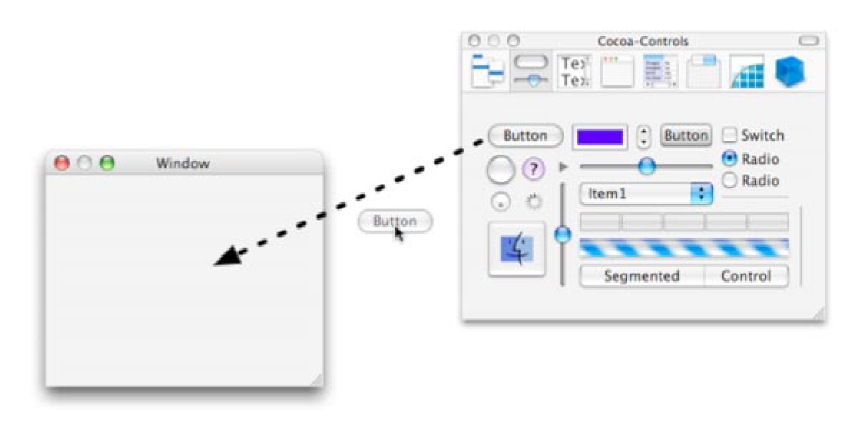


الضغط المتتالي لفتح الملف MainMenu.nib

يؤدي الضغط المتتالي على MainMenu.nib داخل Xcode لتشغيل برنامج آخر يدعى Interface Builder واختصاره IB، حيث سيظهر لك نوافذه الخاصة، يمكنك عندها اخفاء نوافذ التطبيقات الاخرى حتى تتمكن من التركيز على ما يتيحه لك Interface Builder.

يعرض Interface Builder عدد ٣ نوافذ احداها تسمى "Window" والتي ستكون النافذة الظاهرة لمستخدمي برنامجك، انها كبيرة تقريباً، يمكنك تحجيمها. والى يمينها يوجد نافذة يبداء اسمها بـ "Cocoa-" . وهي عبارة عن مخزن لكافة انواع الكائنات البصرية المتاحة وهذه النافذة تعرف باسم "palettes window" .

اضغط الايقونة التالية من شريط الأدوات Tool Bar باعلى مخزن الكائنات، وقم بسحب الايقونة التي تمثل ازرارين اسفل ذلك الشريط ثم قم بإفلاتها على النافذة الرئيسة المسماة "Window" .اضغط الايقونة التالية من شريط ادوات مخزن الكائنات واسحب الكائن الممثل لحقل نص text field وكما فعلت سابقاَ قم بسحبها وافلاتها الى النافذة الرئيسة. ستجد عندها ان حقل النص قد اضاف سلسة الحروف التالية "System Font Text" الى النافذة الرئيسة.



سحب الكائنات من المخزن الى نافذة التطبيق.

خلف الكواليس، تؤدي عملية السحب والافلات للكائنات –الأزرار من مخزنها الى النافذة–، تؤدي إلى إنشاء كائن ازرار جديد على النافذة. وكذلك الحال مع حقل النص واى كائن آخر قابل للسحب والافلات.

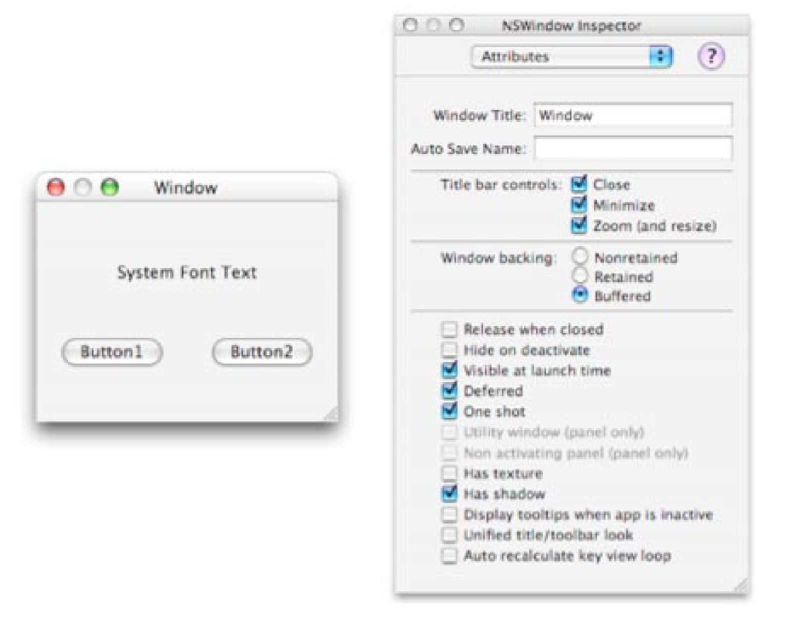
لاحظ ظهور تعليق مقتضب بمجرد ايقاف المؤشر على احد تلك الكائنات، تعليقات مثل NSButton أو NSTextView . تلك التعليقات تمثل اسماء التصنيفات التي تمثلها والتي اعدتها لك أپل، وهي ضرورية لتأدية عدد من العمليات داخل التطبيق.

قم بترتيب موضع تلك الكائنات داخل نافذتك. وتحجيم ما يلزم. قم بتغيير نصوص الازرار، وذلك من خلال الضغط المتكرر على الازرار.

ندعوك لتفحص نافذة مخزن الكائنات حتى تتمرس على عملية اضافة الكائنات إلى النافذة.

لتغيير خصائص properties احد الكائنات، اختر الكائن المراد تغيير خصائصة وذلك بالضغط على المفاتيح command-shift-i. نصحك بتفحص تلك الخصائص ايضاَ.

كمثال اختر النافذ الرئيسة "Window" (سوف يتم اعلامك بذلك مباشرة من خلال وجود تحديد حول مستنسخ النافذة المبين بالنافذة السفلية اليسرى) قم بالضغط على المفاتيح command-shift-i. تفحص نافذة الخصائص واختر "ملامح Attributes" من القائمة المنسدلة pop-up menu. يمكنك تفعيل ملامح النقشات Textured Window التي توفرها النافذة، هذا التفعيل يجعل نافذتك تتخذ الشكل المعدني لاظهار ذاتها. ذلك يعني امكانية تخصيص العرض البصري لعدد كبير من اجزاء برنامج دون الحاجة لكتابة اي سطر برمجي!



دتفحص نافذتنا من خلال برنامج IB.

كما وعدناك سابقاَ،نحن على وشك إنشاء تصنيف class . ولكن قبل البدء بذلك دعنا نتفحص ألية عمل تلك التصنيفات .

فلتوفير الجهود البرمجية، من الافضل دائماَ ان نبني على مابناه الاخرون، بدلاَ من البدء من نقطة الصفر لنعيد إختراع الدائرة . فان كنت بحاجة لإنشاء نافذة جديدة بخصائص وسلوكيات خاصة مثلاَ، فان كل ما هو مطلوب منك هو تزويد الشفرة البرمجية التي توصف تلك الخصائص و السلوكيات . ولن تكون ملزماَ لكتابة اي شفرات برمجية لتوصيف عمليات الاغلاق مثلا، لانها موجودةَ .

ان اعتمادك البناء على ما قام الاخرون ببناه يكسبك كافة تلك السلوكيات المبنية مجانا دون ادني جهدَ. وذلك ما يجعل Objective-C مختلفة اكثر عن لغة C الصرفة .

كيف يتم ذلك ؟

في الحقيقة، هناك تصنيف للنافذة باسم NSWindow ،حيث يمكنك ذلك ان تبني تصنيف خاص بك بحيث يستقى (يرث inherits) كافة خصائص التصنيف الاول اي NSWindow. والان لنفترض انك ترغب باضافة سلوك جديد لتصنيفك المبني على تصنيف النافذة. مالذي سيحدث لو تلقى تصنيف نافذتنا الجديد رسالة الاغلاق "close" ؟

فانت لم تقم بكتابة أو بنسخ اي شفرة برمجية تتعامل مع تلك الرسالة . الامر بسيط جداَ هنا، فاذا ما تلقى التصنيف الجديد رسالة ما لتنفيذ إحراء معين لم نقم بادخال اي شفرة برمجية لمعالجته، عندها سيتم توجيه الرسالة بشكل آلي إلى اي تصنيف اعلى منه بسلسلة الوراثة اي التصنيف الذي ورثنا منه خصائص تصنيفنا أو ما نطلق عليه ("superclass") .

عند الضرورة، يتم اعادة توجية طلب تنفيذ الاجراء باتجاه التصنيف الاعلى فالاعلى، حتى يتم ايجاد ذلك الاجراء وتنفيذه من اعلى الهرم المكون منه هذا التصنيف (top of the hierarchy of inheritance) .

> في حال لم يتم العثور على ذلك الاجراء نكون عندها قد ارسلنا رسالة غير قابلة للمعالجة. انه مثل طلبك لعمال ورشة ان يغيروا دواليب زلاجتك الثلجية -التي لاتحتوي اي دواليب-. في هذه الحالة حتى مدير الورشة لن يستطيع ان يقدم لك شيء. تقوم Objective-C في مثل هذه الحالات باصدار رسالة خطاء.

ماذا لو رغبت بتعديل احد السلوكيات أو الاجراءات الموروثة من تصنيفك الاب في تصنيفك الحالي؟

ذلك امر يسير، حيث يمكنك دوماَ تكييف -override- تلك الاجراءات. كمثال يمكنك اعادة تكييف الاجراء **close** ، الذي يجعل النافذة تخفي ذاتها بصرياَ بحيث يكون إجراء الاغلاق عبارة عن تنحية وتغيير لموقع النافذة ثم اغلاقها. ان التصنيف الكائني لنافذتك يستخدم ذات الاسم المعرف للاغلاق والذي اعدته لك أپل. لذا ما ان يتم توجيه رسالة إجراء الاغلاق لنافذتك، فانها سوف تتنحى وتغير موقعها من الشاشة ثم تغلق ذاتها.

ان عملية اغلاق النافذة لذاتها إجراء مبرمج مسبقاَ من قبل أپل. فمن داخل نطاق تعريفنا لذات الاجراء **close method** قمنا باستدعاء الاجراء الاصلي للاغلاق وهو الموروث من النموذج الاب **superclass** للنافذة. هناك تغييرات طفيفة مطلوبة حتى نضمن الايكون هناك تعاودية **recursive** اثناء استدعاء الاجراء.

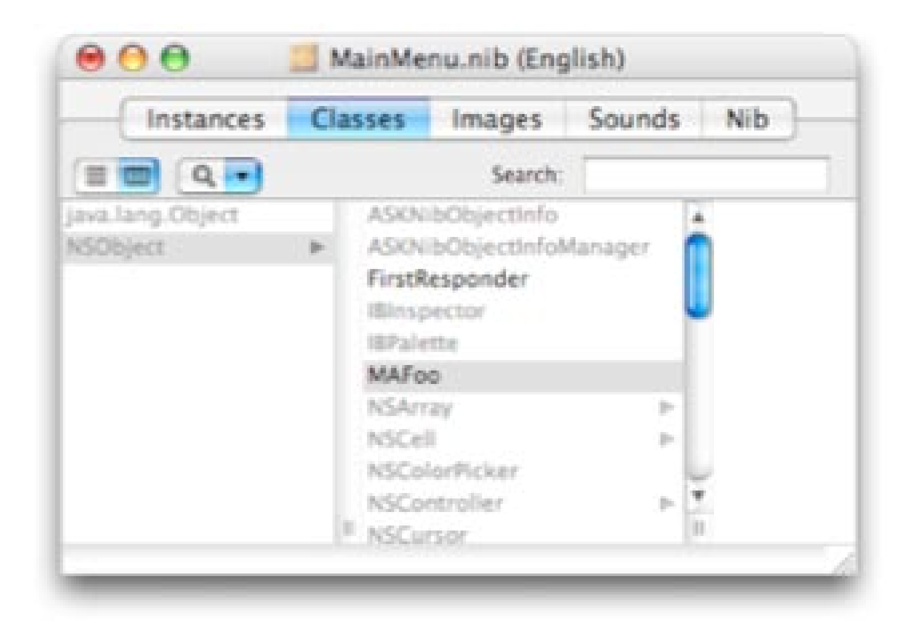
```
[2]
// Code to move the window out of sight
here.
[super close]; // Use the close method of
the superclass.
```

هذا المفهوم البرمجي متقدم جداَ لان يوجد بكتاب للمبتدئين، لذا لن نتوقع منك ان تستوعبه ببضعة اسطر خالية من الدسم.

بالنسبة للكائنات البرمجية، يُعد **NSObject** اصل نشأة ووجود باقي الكائنات فمنه يتحدر الجميع ومنه يستقون توصيفاتهم، وجميعهم يحتفظون بخصائصه وسلوكياته وراثياَ. انه التصنيف الاب لكافة التصنيفات **subclasses** التي ترث منه سواء بشكل مباشر أو غير مباشر. كمثال نجد ان التصنيف **NSWindow** تصنيف متحدر من سلالة التصنيف **NSResponder** وهذا الاخير متحدر من سلالة التصنيف الاب **NSObject**.

ان التصنيف الاب **NSObject** يوصف افعال متجانسة ومعممة داخل الكائنات التي تستقي/ترث منه (كإنشاء نص يصف نوع الكائن، أو قابلية اجراء استفسار ما اذا كان الكائن قادر على تلقي رسالة ما ..الخ) قبل ان اضجرك بالعديد من النظريات، دعنا نرى كيف يُنشأ التصنيف **create a class**.

توجة الى نافذة **MainMenu.nib** واختر بند التصنيفات **Classes**. ستجد رأس الهرم **NSObject** في العمود الاول. اختر هذا التصنيف ثم توجه الى القائمة العليا **menubar** حيث تجد قائمة **Classes menu**. ومن هناك اختر تفعيل امر **Subclass NSObject** ، هذا الامر سينشيء تصنيف يرث صفات النموذج الاب. قم بالعودة مرة أخرى الى **MainMenu.nib**، واطلق على التصنيف الجديد اسماَ ذي دلالة، عن نفسي اطلقت على التصنيف الجديد اسم **"MAFoo"**.



إنشاء تصنيف "MAFoo class"

ستلحظ وجود حرفين كبيرين Capital ضمن اسم مصنفنا، انهما اختزال ابتكرناه للاشارة الى "My Application". يمكنك اختراع ما تراه مناسباً من الاسماء لمصنفاتك. قبل ان تبدأ العمل بكتابة شفرة برنامجك ننصحك بان تتخذ منهج تسمية مشابه. (بمعنى اختيار حرفان أو ثلاثة لتمييز مصنفاتك وتجنب ازدواجية تصادم المسميات مع المصنفات الموجودة أو التي يتم استعارتها وتضمينها من قبل مبرمجين آخرين). على كل حال تجنب استخدم بادئة الحرفين NS ذلك انها بادئة معتمدة لتصنيفات أپل . وهي اختزال لاسم NextStep. فـ NextStep كان النظام الاساسي الذي يقوم عليه نظام Mac OS X والذي قامت أپل بشراءه، وحصلت على ستيف جوبز كهدية اضافية فوق العرض.

موقع دائرة المعلومات المتشاركة CocoaDev wiki يحوي قائمة بالحروف البادئة prefixes التي يجب تجنبها. يجدر بك زيارة الموقع متى ما قررت اعتماد بادئة تسميات خاصة بك.

```
http://www.cocoadev.com/index.
pl?ChooseYourOwnPre x
```

متى قمت بإنشاء تصنيف جديد، احرص على اعطاءه اسماً ذي دلالة ومعنى موجز عن دوره. كمثال وكما رأينا في تصنيفات Cocoa حيث كان تصنيف النافذة متخذاً الاسم NSWindow . مثال آخر عن تصنيف يختص بإظهار الالوان وهو تحت اسم NSColor . في حالتنا يعد اسم التصنيف MAFoo الذي قمنا بإنشاءه مجرد مثال لشرح الطريقة التي سيتعامل بها البرنامج مع هذا التصنيف. لذا اطلقنا عليه هذا الاسم العبثي دون الدلالة لاي دور يقوم به.

بالعودة الى برنامج IB ، قم باختيار إستنساخ Instantiate لعمل مستنسخ من تصنيف MAFoo الذي ستجده قد اضيف داخل

قائمة التصنيفات Classes menu. ،كما سترى ادنى لويحة المستنسخات Instances tab، لديك الان ايقونة جديدة باسم MAFoo. هذه الايقونة تقوم مقام المستنسخ الجديد الذي قمت بإنشاءه.

إنشاء مستنسخ MAFoo

خطوتنا التالية حول إنشاء توصيلات ما بين الازرار (التي ستقوم بارسال الرسالة) الى كائن MAFoo (الذي سيتلقى تلك الرسائل).
بالاضافة الى ذلك، سنقوم بجعل كائن MAFoo قادراً على ارسال رسائل موجهة منه شخصياً الى حقل النص، ذلك ضروري لان حقل النص كائن يستقبل الرسائل.

ان اي كائن غير قادر على التواصل مع الكائنات الاخرى في ظل عدم وجود توصيلة مرجعية **reference** تدله على الكائن المستقبل.

عندما نقوم بالتوصيل ما بين الازرار ومصنفنا الكائن MAFoo، فإننا بذلك نزود الازرار باشارة مرجعية كجسر تواصل تشير للكائن MAFoo. الازرار هنا سيتمكن من ارسال الرسالة الى الكائن MAFoo. وبالمقابل كائننا MAFoo سيحتاج لمد جسر التواصل مع كائن حقل النص، حتى يتمكن الاخير من تلقي الرسالة والامتثال لها.

دعنا نواكب ما يقوم به البرنامج مرة اخرى. ان كل ازرار من الازارير هنا قادر على ارسال رسالة لحدث فريد كرد فعل مثلاً لضغطة مؤشر الماوس الذي تم على ذلك الازرار. متى تم ضغط الازرار قام الاخير بارسال رسالة بناء على ذلك الحدث. هذه الرسالة تحتوي مايلي:

– اسم الاجراء المراد تنفيذه من التصنيف MAFoo الذي سيقوم فعلياً بتنفيذ الامر.

– هذه الرسالة توجه الى الكائن المستنسخ من التصنيف MAFoo الذي أنشاءناه مؤخراَ.

في الحقيقة الكائن المستنسخ MAFoo ، لايوجد لديه اي خبر حول كيفية تنفيذ الامر المرسل بالرسالة. لكن التصنيف MAFoo لديه الخبر اليقين

لذا ما يجري هنا هو ان يتم توجية الرسالة الى الكائن المستنسخ MAFoo والذي بدوره سيقوم برفعها الى موصف تصنيفه اي MAFoo ومن هناك سيتم توجية تلك الرسالة الى كائن حقل النص.

مثلها مثل اي رسالة، هذه الرسالة مكونة من اسم الاجراء method، وهو في هذه الحالة الاجراء الذي يأمر حقل النص بعرض القيمة في اطاره. ان القيمة المرسلة تعد جزء من تكوين الرسالة، بالاضافة الى الاسم الذي سينّشط حقل النص لتغيير محتواه.

لذا تصنيفنا MAFoo بحاجة لتعريف إجرائين يطلق عليهما actions حيث سيتم استدعائهما من خلال كائني الازرارين.

وتصنيفنا MAFoo بحاجة لتعريف منفذ خرج واحد (outlet)، ومنفذ الخرج عبارة عن متغير يحتفظ بعنوان الكائن الذي سيتم التواصل معه (في حالتنا هذه حقل النص هو ذلك الكائن) وذلك من خلال ارسال الرسائل اليه.

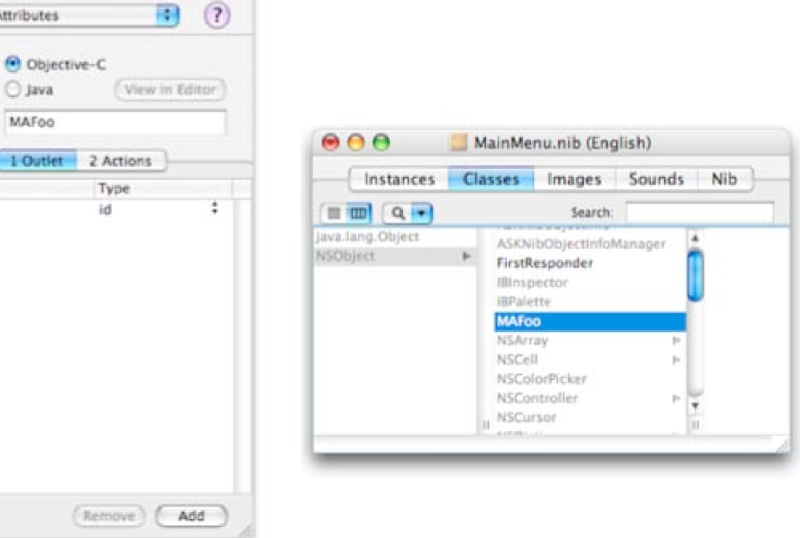
تيقن من اختيارك للتصنيف MAFoo بلويحة التصنيفات Classes tab بالنافذة ذات العنوان MainFile.nib.

ومن خلال لوحة المفاتيح اضغط المفاتيح التالية command-shift-i حتى تظهر لك نافذة تدقيق الخصائص الخاصة بهذا التصنيف.

في نافذة تدقيق الخصائص، اختر لويحة الافعال Action tab واضغط الازرار المعنون بـAdd حتى يمكنك اضافة الاجراء (يطلق عليه Action أو method وكلاهما نفس المعنى) الى تصنيفنا .

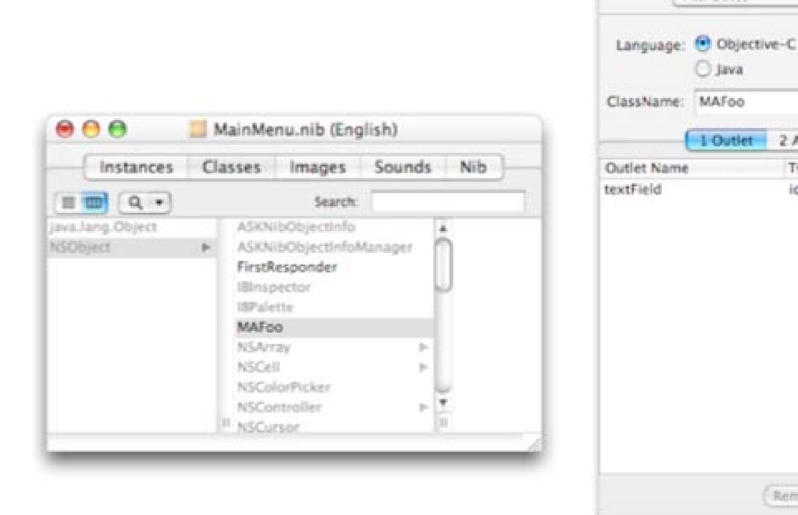
قم بتغيير الاسم الافتراضي للإجراء المضاف باسم ذي دلالة وصفية مثل "setTo5:" –ذلك اننا سنبرمج الفعل ليعرض العدد ٥ كنص داخل كائن حقل النص– ثم قم باضافة إحراء آخر وقم بتسميته "reset:" – على سبيل المثال ذلك اننا سنبرمج هذا الاجراء ليعرض قيمة صفر بحقل النص–.

ولاحظ ان كافة الاجراءات المستحدثة هذه تنتهي بنقتطين عموديتين(":"). المزيد من المعلومات لاحقاَ.



إضافة الاجراءات action methods للتصنيف MAFoo

الان وانت بنافذة تدقيق الخصائص، اختر لويحة الخرج Outlet tab، وقم بإضافة مخرج outlet واطلق عليه اسم "textField" مثلاَ.



إضافة مخرجات outlet للتصنيف MAFoo

قبل البدء بعمل التوصيلات اللازمة بين الكائنات، يجدر بنا اعطاء الازرارين اسماء ذات دلالة. وبما ان الازرار الاول سيقوم بالطلب من مستنسخ MAFoo ان يعرض العدد ٥ بحقل النص، لذا سنقوم بتسمية الازرار "Set to 5" –نحن نعلم كيفية عمل ذلك من خلال الضغط المتتالي على الازرار ثم تغيير النص الى الاسم الجديد–. كرر الامر ذاته مع الازرار الثاني، حيث سنطلق عليه اسم "Reset" .

لاحظ ان عملية تغيير مسميات العناوين التي اجريناها تعد عملية غير اساسية لعمل برنامجنا. انها مسألة تتعلق بتعديل الواجهة الرسومية للكائنات حتى تكون ذات دلالة وصفية اكبر للمستخدم.

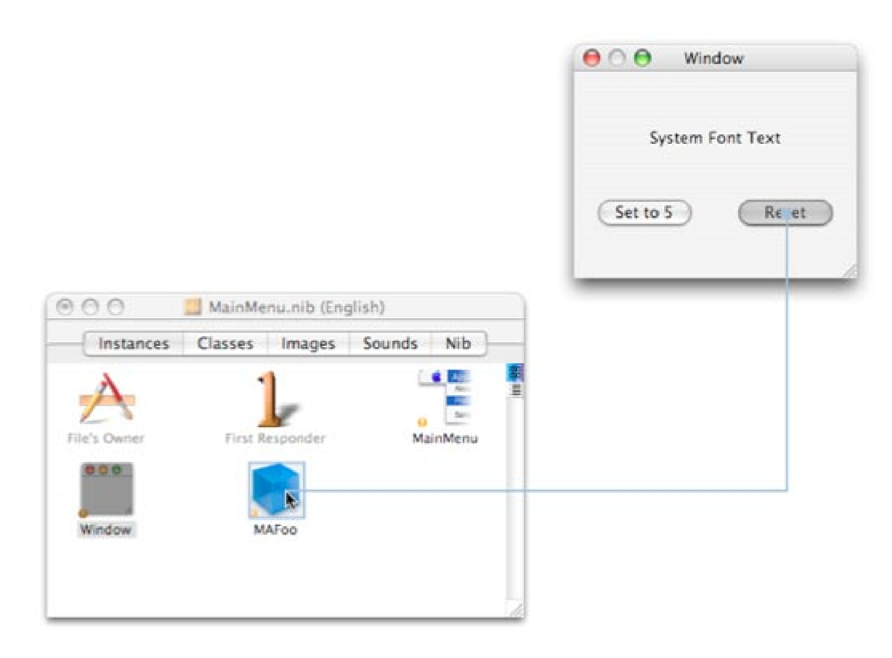
نحن الان جاهزون لمد جسر التواصل بين الكائنات حيث سنعمل التوصيلات اللازمة بين كل من:

١) توصيل الازرار"Reset" بمستنسخ MAFoo

٢)توصيل الازرار"Set to 5" بمستنسخ MAFoo

٣) توصيل المستنسخ MAFoo بحقل النص

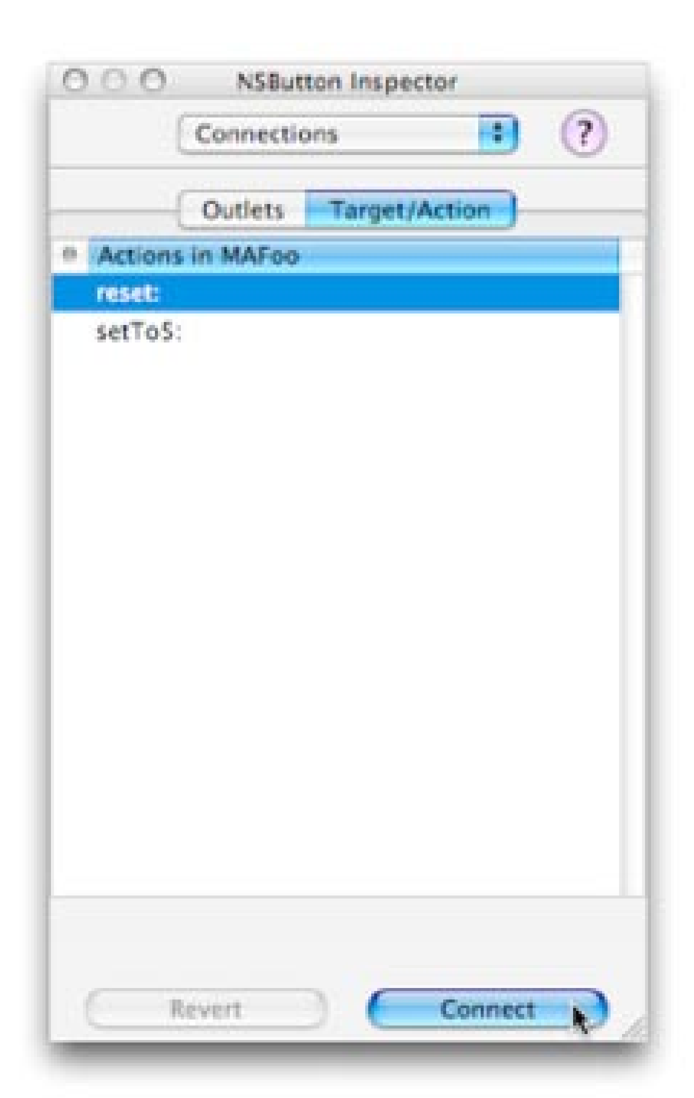
لعمل هذه التوصيلات، اضغط لويحة المستنسخات Instances tab بنافذة MainFile.nib. ثم بالضغط على مفتاح Control الموجود بلوحة المفاتيح استخدم الفأرة لعمل سحب وافلات منطلقاً من ازرار Reset الى مستنسخ MAFoo (وانتبه للأ تقم بعكس هذه التوصيلة اي ان تنطلق من المستنسخ وتنتهي بالازرار!). عندها سيظهر خط بين نقطة الانطلاق ومكعب صغير عند الانتهاء، هذا الخط يمثل التوصيلة التي قمت بعملها. اجعل الخط منطلقاً من الازرار ودعه ينتهي عند مستنسخ MAFoo وعندها افلت ضغطك من الفأرة.



توصيل الازرار بكائن MAFoo

عندما قمت بافلات الضغط عن الفأرة، قام محقق خصائص الازرار بعرض التوصيلات في قائمة تعرض الاجراءات action methods التي تمثل ما يقدمة الكائن MAFoo.

قم باختيار الاجراءد المناسب (اعني "reset:") واضغط على الزر Connect لاتمام اجراء عملية التوصيل.



نفعيل التوصيلات من خلال المحقق inspector.

الان سترى ان الازرار يحتفظ بعناوين مرجعية موجهة الى الكائن MAFoo. هذا الازرار سيقوم بتوجية رسالة الى ذلك الكائن MAFoo كلما تم ضغطه. يمكنك الان اجراء التوصيلات التي تربط الازرار Set to 5 بالكائن MAFoo من اخلال اعادة نفس الاجراء السابق.

ولعمل توصيلات مابين الكائن MAFoo وحقل النص، ابداء اولاَ بالكائن MAFoo ثم اضغط مفتاح control وقم بعملية سحب منطلقاَ إلى كائن حقل النص، اضغط الازرار Connect وبذلك تكون انتهيت من عملية التوصيل.

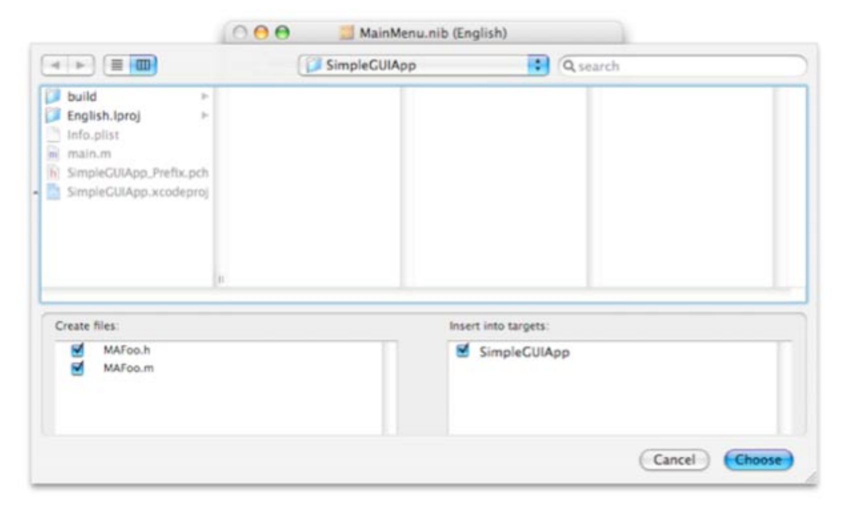
ترى ما كل هذا؟ ولماذا؟

في الحقيقة وكما سترى بعد قليل، انك قد قمت باعمال جنبتك عن كتابة أي سطر برمجي، فما قمت به قبل قليل يمكنك عمله من خلال عدد لابأس به من اسطر الشفرة البرمجية.

تأكد من ان مستنسخ MAFoo قيد الاختيار بنافذة .MainMenu nib ثم قم بتحويل عرض البيانات بالضغط على لويحة التصنيفات Classes tab. ويفترض انك ترى الان لائحة بكافة التصنيفات وسترى ان تصنيفنا MAFoo قيد الاختيار حالياَ.

اختر قائمة Classes menu الموجودة بالقائمة العليا، من ثم فعّل

امر إنشاء ملفات التصنيف Create Files for MAFoo . عندها سيسألك IB عن الموقع الذي سترغب بحفظ ملفاتك قيد الانشاء. افتراضياَ هذه الملفات سوف تنشأ داخل مجلد المشروع، وهذا العنوان هو بالضبط ما نريده.



إنشاء الشفرة البرمجية للتصنيف MAFoo

الان قم بالعودة الى برنامج Xcode، وسوف ترى ان الملفات التي أنشأتها موجودة داخل نافذة المشروع، بمجلد المصادر الاخرى Other Sources group . حيث يمكنك سحب هذه الملفات وادراجها بمجلد التصنيفات Classes group في حال الرغبة، فكما ترى ان هذه الملفات الحديثة الإنشاء تمثل توصيف التصنيف MAFoo.

استعراض الملفات المنشاءة داخل Xcode.

دعنا نعود للوراء قليلاَ، حيث الفصل ٤، عندما ناقشنا وظيفة الدالات. هل تتذكر نقاشنا حول ترويسة الدالة function header بالمثال [11.1]؟ لقد كانت بمثابة تصريح اخباري للمركم، تخبرة عما قد يصادفة من اعلانات وتعاريف عن الدوال وانواع المعاملات.

إن احد ملفاتنا المنشاءة مؤخراَ يحمل الاسم MAFoo.h وترويسة ذلك الملف تحوي المعلومات التي توصّف هذا التصنيف. ففي

السطر [3.5] تجد كلمة **NSObject** المألوفة لديك، والتي تخبرنا وتخبر المجّمع ان هذا التصنيف يستقي / يرث صفاته من التصنيف الاب **NSObject class** .

```
[3]
/* MAFoo */

#import <Cocoa/Cocoa.h>  // [3.3]

@interface MAFoo : NSObject
{
    IBOutlet id textField; // [3.7]
}
- (IBAction)reset:(id)sender;
- (IBAction)setTo5:(id)sender;
@end
```

سترى ايضاَ [3.7] وجود منفذ خرج **outlet** يستخدم حقل النص كمعامل –الجديد هنا هو"**id**" وهي تعني كائن، أو في الحقيقة هي مؤشر للكائن حيث تنوب عنه هنا –.

ان "IB" تشير الى **Interface Builder** ، ذلك البرنامج الذي استخدمناه لإنشاء هذه الشفرة البرمجية.

ستجد ايضاَ كلمات **IBAction** في [3.9, 3.10] وهي مماثلة للدالات التي لاتقوم بإرجع اي قيمة **void**. ودورها هنا يتمثل بارسال الرسالة: فقط و التي لا تتطلب اي رد يليها عند ارسالها الى الكائن **MAFoo** .

سترى ايضاَ عدد اثنين من الاجراءات التي يستخدمها **IB**.

لقد رأينا سابقاَ مفردة **#import <Foundation/Foundation.h>** بدلاَ من السطر[3.3] . فالمفردة الاولى تستخدم ما يلزم من دوال ووظائف للبرامج التي لا تعتمد إستخدام إمكانيات ووظائف واجهة التطبيقات الرسومية او **GUI**، بينما ما تراه الان بالشفرة البرمجية يستنفذها استنفاذاَ .

دعنا الان نتفحص الملف المُنشاء الثاني . انه الملف المسمى **MAFoo.m** . ومرة اخرى حصلنا على سطور وشفرة برمجية مجاناَ دون اي تدخل أو عناء من قبلنا.

```
[4]
#import "MAFoo.h"

@implementation MAFoo

- (IBAction)reset:(id)sender // [4.5]
{
}

// ...Continued
```

```
- (IBAction)setTo5:(id)sender
{
}

@end
```

قبل اى شيء، سوف يتم جلب تصريحات ترويسة الملف MAFoo.h و ذلك حتى يأخذ المجّمع فرصتة في التعرف عما سيواجهه.

يوجد مفردتان مألوفتان لنا هنا انها **:reset** و **:setTo5** . هذه الاوامر هي نفسها الاجراءات **methods** التي حددناها للتصنيف.

انها شديدة الشبه بالدالات ذلك انها تتطلب وضع الاوامر محصورة بين الاقواس المعقوفة. ففي برنامجنا متى ما تم ضغط احد الازرار، سيقوم الازرار المضغوط بارسال رسالة الى الكائن **MAFoo**، تطلب من تنفيد احد تلك الافعال **methods**. الجميل في الموضوع اننا غير ملزمين بكتابة اي شفرة برمجية لتحقيق ذلك.

ان ما قمنا به من اعمال توصيل بين الازرار والكائن **MAFoo** من خلال **IB** كان هو كل المطلوب، وقد جنبنا ذلك كتابة هذه السطور التي تراها الان. على كل حال علينا كتابة الشفرة البرمجية التي ستجعل الكائن **MAFoo** يرسل رسالته إلى حقل النص ليقوم الاخير يتغيير القيمة المعروضة كما هو موضح بالاسطر [5.7, 5.12] .

```
[5]
#import "MAFoo.h"

@implementation MAFoo


- (IBAction)reset:(id)sender
{
  [textField setIntValue:0]; // [5.7]
}


- (IBAction)setTo5:(id)sender
{
  [textField setIntValue:5]; // [5.12]
}


@end
```

كما ترى نحن نرسل الرسالة الى ذلك الكائن المخزن بمنفذ الخرج **outlet** الخاص بحقل النص **textField**. وبما اننا قد قمنا باعداد التوصيل مابين منفذ الخرج وكائن حقل النص باستخدام برنامج **IB**، لذا سيتم توجيه الرسالة الى وجهتها الصحيحة.

ان اسم الاجراء **method** المستخدم لعملية ارسال الرسالة هو **:setIntValue** وهو يتطلب معامل عددي صحيح **Integer**. كما

ان إجراء :setIntValue قادر على اظهار القيمة المرفقة معه في كائن حقل النص.

في الفصل التالي سوف نخبرك كيفية اكتشافنا لهذا الايعاز. انت الان جاهز لتجميع تّجميع البرنامج وتشغيله. وكما هو معتاد قم بالضغط على الازرار **Build** الموجود بشريط الادوات، سيتطلب الامر عدة ثوان لبناء وتشغيل البرنامج. في النهاية سينطلق البرنامج وتظهر نافذته على الشاشة حتى تتمكن من تجربتة واختباره.



برنامجنا قيد التشغيل.

باختصار لقد قمت بإنشاء برنامج بدائي بسيط، وقد تتطلب منك كتابة سطرين من الشفرة البرمجية!

بحث التعليمات البرمجية

في الفصل السابق تعرضنا لبعض الاجراءات ، وقد قمنا بكتابة اثنان منهما بانفسنا (بُنية الاجراء)، لكننا قمنا باستخدام اجراء واحد جاهز من قبل أپل . فالاجراء **setIntValue:** كان ذلك الامر الموكل بعرض القيمة الرقمية بمحتوى كائن حقل النص. ترى كيف تمكنا من معرفة ذلك؟

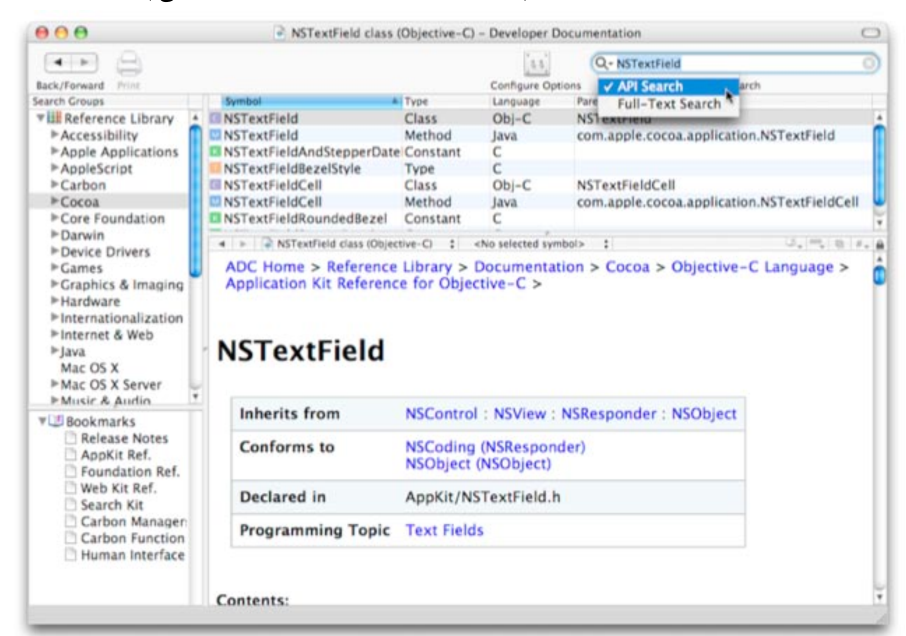
> هل تذكر ان اي اجراء مقدم اليك من قبل أپل لا يتطلب منك كتابة اي شفرة برمجية. بالاضافة الى ذلك هذه الاجراءات تضمن لك خلوها من الاخطاء البرمجية. لذا تقضي الحكمة ان تستثمر بعضاَ من الوقت للتقصي والبحث عن تلك الاجراءات المتوفرة لك قبل ان تقوم بكتابة اي سطر برمجي.

اذا ما قمت باختيار احد الكائنات التي يعرضها لك برنامج IB ستجد توصيف مقتضب يشير لاسم ذلك التصنيف الذي يقف عنده المؤشر، انه توصيف مختصر جداَ مثل **NSButton** .واذا ما اوقفت مؤشر الفأرة فوق كائن حقل النص فانك ستقراء **NSTextField** . ان كل اسم من تلك الاسماء يمثل اسم ذلك التصنيف، دعنا نتفحص **NSTextField** وما يقدمة التصنيف من اجراءات **methods** .

الاستدلال على الايعازات الاستقصاء عن والاوامر

توجه الى Xcode وستجد بالقائمة العليا menu ، بند قائمة المساعدة باسم Help->Documentation ،

اختر Cocoa من الاطار الايسر ثم ادخل مفردة NSTextField بحقل البحث ، تأكد من اتاحة خيار البحث ان يشمل اجراءات واجهة التطبيقات API-Search، (انظر للصورة المرفقة بالاسفل) .



التجوال بين وثائق Cocoa باستخدام Xcode.

اثناء ادخالاك للمفردة، ستجد ان قائمة نتيجة البحث التي تحوي احتمال وجود تلك المفردة قد بدأت بالتقلص، عندها ستجد تلك المفردة NSTextField قد ظهرت اعلى القائمة . واضغط على السطر الذي يشهر مفردة البحث NSTextField (وهي هنا من نوع تصنيف Class) حتى تحصل على معلومات ذلك التصنيف . وسوف يعرض التصنيف معلوماته بالاطار السفلي .

اول امر نلاحظة هنا هو ان هذا التصنيف مكون بالوراثة من سلسلة متصلة من التصنيفات الاخرى . اخر تصنيف في القائمة هو الاب، أو التصنيف الجذر، انه ملك الهرم NSObject !

انزل للاسفل قليلاَ (scroll) وستجد ان العنوان الرئيس التالي :

```
Method Types
```

إبدء بحثك عن الاجراءات تحت هذا العنوان . ان نظرة سريعة على العناوين الفرعية تدل على اننا لن نجد ذلك الاجراء الذي نحتاجة لعرض تلك القيمة النصية داخل محتوى كائن حقل النص .

يعود سبب ذلك لمبادىء التوارث بين الكائنات، فنحن بحاجة لتفحص التصنيف الاعلى في السلالة superclass الذي تكون منه تصنيف NSTextField ، وهو بطبيعة الحال تصنيف NSControl (وفي حال فشلنا في العثور عن ما نبحث عنه سنتوجه الى

التصنيف الاعلى في السلالة، الا وهو NSView ، وهكذا حتى نصل الى الجذر الاب) .

بما ان وثائق المساعدة التوضيحية مكتوبة باستخدام لغة HTML، فكل ما علينا عمله الان هو الضغط على كلمة NSControl (السلالة المتحدرة مرتبة كما هو موضح). تلك الضغطة تنقلنا الى الصفحة التي تحوي توصيف NSControl.

NSControl

Inherits from	NSView : NSResponder : NSObject

كما ترى لقد انتقلنا خطوة للأعلى داخل سلسلة الوراثة. وفي العناوين الفرعية للاجراءات سنجد التبويب التالي :

```
Setting the control's value
```

أو مجموعة إسنادات القيم للتصنيف، وهذه الاجراءات هي ما نبحث عنه، اننا نرغب باسناد قيمة. لذا اسفل ذلك العنوان الفرعي سنجد :

```
- setIntValue
```

: حتى الان الامور واضحة، لذا سنتفحص وصف ذلك الاجراء من خلال الضغط على الرابط المحيط بكلمة setIntValue: .

الإجراء :setIntValue وصيغته

```
- (void)setIntValue:(int)anInt
```

انه إجراء يقوم بإسناد قيمة عددية anInt داخل خلية (او خلية قيد الاختيار). واذا ما كانت الخلية قيد التحرير، عندها سيتم ايقاف كافة العمليات قبل اتمام عملية الاسناد، ذلك؛ في حال لم تكن الخلية سليلة متحدرة من سلسلة الوراثة للتصنيف NSActionCell.

ان هذا الاجراء يقوم بجدولة عملية تحديث عرض قيمة محتوى الخلية (ان التصنيف NSActionCell يقوم بجدولة اعمال التحديث الخاصة بالخلايا التي يمتلكها).

في برنامجنا نجد ان كائن حقل النص NSTextField هو المستقبل للرسالة وهو بحاجة لادخال قيمة من نوع عددي صحيح في المعامل المستخدم بالاجراء. ويمكننا تبين ذلك بدلالة توقيع الاجراء:

```
- (void)setIntValue:(int)anInt
```

تستخدم Objective-C رمز علامة الناقص "-" في بداية الاجراءات المصرحة كإجراءات تخص الكائن المستنسخ (وهي التصريحات المقابلة بتصريحات التصنيف ذاته، وسنقوم بتغطية ذلك الموضوع لاحقاَ).

ان مفردة void البادئة بهذا الاجراء تعني عدم وجود قيمة مرتجعة. بمعنى ان الاجراء :setIntValue بالرسالة المرسلة الى حقل النص، من قبل المرسل لن تجعل هذا المرسل (MAFood هل تتذكره؟) يستقبل اي قيمة راجعة من كائن حقل النص جراء تنفيذ هذا الاجراء. هذا هو الطبيعي. فبعد النقطين العموديتين ":" نلاحظ ان معامل الاجراء (int) يشترط ان يكون المتغير المرسل عدد صحيح. ففي مثالنا المستخدم، قمنا باسناد قيمة عددية تساوي ٥ أو قيمة تساوي صفر، وكلاهما عدد صحيح، لذا نحن نسير بشكل سليم.

في بعض الاحيان يصعب عليك تحديد الاجراء الانسب للاستخدام. لذا قم باستشارة ومراجعة مكتبة وثائق المساعدة، وتمرن ومارس ما تتعلمه.

مالعمل عندما ترغب بالاستعلام عن القيمة التي يحتويها كائن حقل النص؟

هل تتذكر ذلك المبدء الجميل الذي يحجب عنك المتغيرات الخاصة بنطاق الدالة؟ هذا المبدء ينطبق هنا في حالة الاجراءات. على كل حال غالباَ ما ستجد ان الاجراءات تأتي بشكل توأم متلازم، ندعو تلك الاجراءات المتلازمة بـ "Accessors"، فهناك إجراء خاص بالاستعلام عن القيمة، والاخر يختص باسناد تلك القيمة. ولقد تعارفنا على الاخير الذي يقوم بالاسناد :setIntValue على كل حال توأمه المتلازم سيكون مثل ما يلي:

```
[1]
- (int) intValue
```

كما ترى، هذا الاجراء، يقوم بارجاع قيمة عددية صحيحة. لذا متى ما اردنا الاستعلام عن قيمة تلك القيمة العديدة الصحيحة المضّمنة داخل كائن حقل النص فاننا سنقوم باستدعاء وتنفيذ هذا الاجراء كما يلي:

```
[2]
resultReceived = [textField intValue];
```

مرة أخرى، كما هو الحال مع وظائف الدوال **functions** (والاجراءات كذلك) ستكون جميع المتغيرات الداخلية بتلك الدوال محجوبة ضمن نطاق دوالّها. هذه ميزة جميلة جداَ لانها تحمي اسماء المتغيرات، ولن يعتريك الخوف من اسناد قيمة لمتغير في احد اجزاء برنامجك خوفاَ من يؤدي ذلك الاسناد لتغيير قيمة متغير آخر يصدف ان يكون بنفس الاسم.

تدفع Objective-C ميزة الحجب هذه شوطاَ ابعد، حيث تشترط ان ان تكون اسماء الاجراءات فريدة فقط داخل نطاق التصنيف، لذا

ستجد عدة تصنيفات مختلفة لديها اجراءات باسماء موحدة.

هذه ميزة مهمة تلمس قيمتها متى عملت على تطوير برامج ضخمة، حيث يتشارك في كتابتها عدة مبرمجين ، وكل واحد منهم مسئول عن تصميم التصنيف الذي يراه مناسباَ، دون وجود مشاكل تتعلق بتصادم أو إزدواجية باسماء الاجراءات.

وهناك المزيد من الفائدة. حيث ان حقيقة وجود عدة إجراءات متماثلة الاسماء داخل عدة تصنيفات يطلق عليها تقنياَ **polymorphism** . وهذه الاخيرة هي ما يجعل البرمجة بالكائنات **object-oriented programming** واختصارها **(OOP)** اسلوباَ برمجياَ فريداَ.

فهذه الميزة تمكنك من كتابة كتل من الشفرات البرمجية دون ان يتطلب منك ذلك معرفة مسبقة عن كائن التصنيف أو التصنيفات **classes** التي تتعامل معها.

ذلك كل ما هو مطلوب، فعند تشغيل البرنامج فعلياَ **run-time** ستقوم تلك الكائنات بتفهم واستيعاب الرسائل الموجهة اليها. ان الاستفادة من هذه الميزة، تمكنك من كتابة تطبيقات قابلة للتعديل والتطوير.

كمثال، نورد برنامجنا الذي استخدمنا فيه واجهة التطبيقات الرسومية **GUI**. ففي حال اردنا استبدال كائن حقل النص، بكائن آخر من فصيلة مختلفة ولكن الاخير يقبل تنفيذ الإجراء **setInValue:** ، عندها سيظل برنامجنا يعمل دون الحاجة لتغيير اي شفرة برمجية، أو حتى الحاجة لاعادة تركيب البرنامج من خلال المجّمع.

نحن قادرون حتى على تغيير نوع الكائن وقت التشغيل دون ان يكون لذلك ضرر على اي قطاع من البرنامج على الاطلاق.

هنا تكمن القوة والفائدة عند استخدام البرمجة بالكائنات (OOP) .

الاستنهاض من الغفوة

قامت أپل بعمل الكثير والكثير لتسهل عليك إنشاء برامجك. ففي برنامجنا الصغير، لم نكترث لبرمجة النافذة كي ترسم ذاتها على الشاشة، ولم يكن مطلوباً منا كتابة اي سطر برمجي لنشرح للإزرار كيف يرسم ذاته على النافذة، بالاضافة الى امور وتفاصيل اخرى. فهذه الاعمال من مسئولية هيكلي عمل frameworks ، اثنين هما Foundation Kit framework و .Application Kit framework

الاول Foundation Kit، هو والذي تعاملنا مع وجلبناه في مثالنا [12] بالفصل ٤ ، حيث كانت مهمتة تزويدنا بخدمات ليس لها علاقة بالواجهة الرسومية. والهيكل العملي الثاني Application Kit ، هو المسؤل المباشر عن اي كائنات رسومية تراها معروضة على الشاشة امامك، فهو يحقق العرض والتداول user-interaction mechanisms . دعنا نعود مرة اخرى الى برنامجنا الذي يعتمد واجهة التطبيقات الرسومية GUI، ولنفترض اننا نرغب من برنامجنا ان يعرض قيمة معينة بكائن حقل النص ما ان ينطلق البرنامج للعمل ويبدأ بعرض نافذته.

ان جميع المعلومات التي تهتم لها نافذة البرنامج مخزنة بملف من امتداد nib (و nib هذه تمثل امتداد مختصر لكلمة Next Interface Builder).

ان في ذلك اشارة الى ان الاجراء الذي نحتاجة يختص بقطاع هيكل العمل Application Kit . لذا دعنا نتفحص كيفية حصولنا على المعلومات حول هذا الهيكل العملي . ومن خلال Xcode, توجه إلى قائمة المساعدة Help menu واختار وثائق المساعدة Documentation.

في نافذة وثائق المساعدة تأكد من ان خيار بحث كافة النصوص Full-Text Search قيد التفّعيل (ولعمل ذلك قم بالضغط على ايقونة المكبر بحقل البحث الموجود بتلك القائمة) . ثم قم بكتابة Application Kit بحقل البحث يليها مفتاح الادخال Return.

سيقوم Xcode بتزويدك بعدة نتائج متنوعة . ومن ضمن هذه النتائج ستجد وثيقة باسم Application Kit Reference for Objective-C . داخل الوثيقة ستجد قائمة من الخدمات التي يوفرها هذا الهيكل .

ستجد تحت العنوان الفرعي للقسم Protocols رابط يسمى NSNibAwakening.

الإستنهاض بروتوكل NSNibAwakening

انه إجراء للإعلان والتصريح، يحتوي ما يلي :

NSNibAwaking

Declared in	AppKit/NSNibLoading.h
Programming Topic	Loading Resources

Contents:
Protocol Description
Method Types
Instance Methods

Protocol Description

This informal protocol consists of a single method, awakeFromNib. Classes can implement this method to perform final initialization of state after objects have been loaded from an Interface Builder archive.

البروتوكول عبارة عن طريقة ذات آلية عمل متفق عليها، فهو هنا بروتوكول اعلامي informal، يتكون من إجراء واحد، يتمثل بـ awakeFromNib .

تقوم التصنيفات بإقحام –implement– هذا البروتوكل ليساعدها بتهيئة حالتها النهائية متى ما تم جلبها من ارشيف الملف الذي يخزنها والذي يكون امتداده NIB . فمتى ما اقحمنا -implement- هذا الاجراء، فانه سيكون قيد التنفيذ لاجراء التهيئة المناسبة للكائن

الذي قام باستدعائة وهو في مرحلة التحميل من الارشيف. بناء على ذلك يمكننا استخدامة لتحقيق هدفنا: اي عرض قيمة افتراضية معينة وقت التشغيل.

> نقترح عليك دوماَ ان تجري بحثك وتحوياتك عن الاجراءات الانسب. و غالباَ ما يتطلب ذلك قليلاَ من التصفح والاستخدام الخلاق لمفردات البحث المستخدمة لتسليط الضوء على الاجراء المطلوب.

لذا من المهم جداَ ان تتألف ببحثك خلال تلك الوثائق حول هذان الهيكلان .**frameworks** حتى تتمكن من التعرف على التصنيفات التي يقدمانها، والاجراءات التي سوف تكون تحت تصرفك.

قد تقرأ عن معلومات ليست بذات اهمية لبحثك الحالي، ولكنك ستكتشف انك تبني قاعدة من المعلومات بمخيلتك سوف يكون لها الاثر في سرعة بنائك لبرامجك.

جميل جداَ، لقد وجدنا الاجراء بعد تسليط الضوء عليه، وكل ما علينا عمله الان هو إضافتة إلى ملف توصيف التصنيف **MAFoo.m** كما يلي:

```
[1]
#import "MAFoo.h"

@implementation MAFoo

- (IBAction)reset:(id)sender
{
  [textField setIntValue:0];
}

- (IBAction)setTo5:(id)sender
{
  [textField setIntValue:5];
}

- (void)awakeFromNib // [1.15]
{
  [textField setIntValue:0];
}

@end
```

متى ما قامت النافذة بعرض ذاتها، سيقوم الاجراء **awakeFromNib** بإستدعاء ذاتة إلياَ. وكنتيجة لذلك، سيُستنهض حقل النص مبكراَ ليعرض قيمة تساوي صفر ما ان يكون ظاهراَ لك داخل تلك النافذة المفتوحة.

المؤشرات

من واجبنا تحذيرك ان هذا الفصل يحتوي على عدد من المفاهيم المتقدمة حول اساسيات لغة C والتي قد تُشعر المبتدئين بالضيق والتضجر.

لا تقلق اذا لم تستوعبها الان ولا تجعل الامر يقلقك حيث انه ليس متطلب رئيسي لبدء البرمجة بلغة **Objective-C** ، لكن التعامل مع المؤشرات له فوائد جمة، فعندما تقوم بتعيين وتعريف متغير **variable**، سيقوم الماكنتوش بحجز حيز معين من الذاكرة وذلك لحفظ القيم التي سيحتويها هذا المتغير. كمثال على ذلك، تفحص الايعاز التالي:

```
[1]
int x = 4;
```

حتى يتم تنفيذ هذا الايعاز، سيقوم الماكنتوش بالبحث عن مساحة غير مستخدمة بالذاكرة. ثم سيقوم بحجز حيز منها حيث يتم تخزين قيمة ذلك المتغير x، (وبالتأكيد لك مطلق الحرية في اختيار اسم المتغير).

تفحص الايعاز مرة أخرى [1]، وستجد انه يحدد نوع/صنف المتغير (وهو عددي صحيح هنا)، حتى يتمكن الحاسب من

تحديد المساحة المطلوبة للحجز والاحتفاظ بقيمة المتغير x. لو كانت القيمة من نوع عددي كبير أو عدد مزدوج (long long or double)، سيتطلب ذلك من الحاسب ان يقوم بحجز حيز اكبر من الذاكرة المتاحة.

إن ايعاز الاسناد "x = 4" سيقوم بتخزين قيمة العدد ٤ في ذلك الحيز المحجوز من الذاكرة. بالتأكيد سيتذكر الحاسب اين قام بتخزين قيمة المتغيرx من الذاكرة، أو بمعنى آخر إنه يعرف "عنوان المتغير x من الذاكرة، بهذه الطريقة، كلما استخدمت المتغير x في برنامجك، سيقوم الحاسب بالبحث في المكان الصحيح (العنوان الصحيح بالذاكرة) لايجاد القيمة المخزنة الخاصة بالمتغيرx.

المؤشر (pointer) عبارة عن متغير تكون قيمته "عنوان" لمتغير آخر.

متى ما كان لديك متغير، يمكنك ان تحصل على عنوان موقعه بالذاكرة من خلال اضافة الرمز & كبادئة اسمية.

لذا حتى نتحصل على عنوان المتغير x، يجب علينا كتابة &x.

```
&x
```

عندما يقوم الحاسب بتقييم الايعاز x، فانه سيقوم فوراً بارجاع قيمة ذلك المتغير (في مثالنا السابق كانت قيمة المتغير ترجع القيمة العددية ٤). وعلى النقيض من ذلك عندما يجري تقييم الايعاز &x، فإن الحاسب هنا لن يعود لنا بالقيمة المحفوظة للمتغير، بل سيعود بالعنوان الذي يوجد به ذلك المتغير من الذاكرة. ان العنوان عبارة عن رقم يمثل جزء معين من ذاكرة الحاسب (مثل الارقام التي تمثل غرف الفندق – وفندقنا هنا مكون من آلاف الآلاف من الغرف) سيتم التصريح عن المؤشر كما يلي:

```
[2]
int *y;
```

هذا الايعاز يصرح عن ان قيمة المتغير y تحتوي عنوان من الذاكرة لمتغير آخر من نوع عددي صحيح int. ولنقوم بحفظ عنوان المتغيرx داخل المتغير y (نسند عنوان x الى y تقنيناً) نكتب ما يلي:

```
[3]
y = &x;
```

مع وجود المؤشر، يمكنك الوصول الى قيمة ذلك المتغير الذي يشير اليه، من خلال اضافة علامة معامل الضرب * كبادئة امام اسم المتغير (المؤشر). كمثال، ان تقييم هذا الايعاز

```
*y
```

سيعود بقيمة عددية تساوي ٤. فالايعاز هذا يماثل الايعاز المستخدم لتقييم "x". إن تنفيذ الايعاز

```
*y = 5
```

سيكون مماثلاَ لتنفيذ الايعاز

```
x = 5
```

للمؤشرات فوائدها فقد ترغب في بعض الاحيان ان تشير إلى عنوان احد المتغيرات دون الاكتراث للقيمة التي يحتويها ذلك المتغير، ان تعين متغير ينوب عن دلك المتغير وقت توجيهك للإيعازات. كمثال، لنفرض انك بصدد إنشاء دالة تقوم وظيفتها بزيادة قيمة عددية مقدارها ١ الى احد المتغيرات، وهي تستوجب معامل يحتوي عنوان ذلك المتغير. وسبب ذلك انها ستقوم بتعديل محتوى القيمة المخزنة، وليس فقط استخدامها. لذا يتوجب عليك استخدام معامل من نوع مؤشر "pointer as argument"

```
 [4]
void increment(int *y)
{
  *y = *y + 1;
}
```

يمكنك بعد ذلك استدعاء الدالة لتنفيذ ما تقوم به كما يلي:

```
[5]
int x = 4;
increment(&x);
// now x is equal to 5
```

سلاسل الحروف النصية

حتى الان تعارفنا على عدة انواع من البيانات مثل: **integer, long, float, double, BOOL** ، وقد تعارفنا مؤخراَ على المؤشرات او مايعرف بـ **pointers**.

وقد تعرضنا سطحياَ لسلاسل الحروف النصية، من خلال حديثنا اثناء استخدامها بالدالة **NSLog()**. تلك الدالة التي تمكننا من طباعة سلسلة من الحروف على الشاشة، مستبدلين بعض المعاملات بواسطة الحروف الخاصة كـ "\" أو "%" وعلامة **%d** بقيم متنوعة.

```
[1]
float piValue = 3.1416;
NSLog(@"Here are three examples of strings
printed to the screen.\n");
NSLog(@"Pi approximates %10.4f.\n",
piValue);
NSLog(@"The number of eyes of a dice is
%d.\n", 6);
```

لم نتحدث عن سلسلة الحروف كنوع من انواع البيانات سابقاَ، والسبب وجيه. وهو ان سلسلة الحروف عبارة عن كائن! وأصله في سلالة التحدر بسلسلة التوارث هو التصنيف **NSString** أو

التصنيف NSMutableString . دعنا نتحدث عن هذه التصنيفات حيث سنبدأ بالتصنيف NSString .

```
[2]
NSString *favoriteComputer;
favoriteComputer = @"Mac!";
NSLog(favoriteComputer);
```

من المحتمل جداَ ان يكون السطر الثاني مفهوماَ، ولكن السطر الاول [2.1] يستحق بعض التوضيح.

فهل تتذكر عندما قمنا بالتصريح عن المتغير المؤشر pointer-variable-، كان علينا حينها تحديد نوع البيانات التي سيشار اليها؟ اليك الايعاز الذي قمنا به للتصريح عن ذلك، من الفصل ١١ [2].

```
[3]
int *y;
```

لقد اعلمنا المجّمع بان متغير المؤشرy يحتوي عنوان لحيز من الذاكرة. حيث توجد قيمة عددية صحيحة مخزنة هناك.

في المثال نحن نخبر المجّمع بان المتغير المؤشر هذا يشير الى عنوان بحيز الذاكرة حيث يوجد كائن من نوع NSString .

لقد استخدمنا مؤشر ليقوم بالاحتفاظ بتلك السلسلة من الحروف. يعود السبب في ذلك لطبيعة Objective-C التي تمكنك من التعامل مع وتغيير قيمة الكائنات من خلال المؤشرات وليس من خلال التعامل المباشر.

حتى الان كل شيء جميل، لكن مابال ذلك الرمز الغريب @ الذي يظهر بين سطور شفرتنا البرمجية من حين لآخر؟

كما تعلم تعد لغة Objective-C امتداد متطور من لغة C الاصلية، فالقديمة لها طريقتها في التعامل مع النصوص وسلاسل الحروف عن الجديدة.

وحتى يمكننا تبيّن النوع الحديث من النصوص و سلاسل الحروف –وهو نوع متطور جداَ–، تقوم باستخدام Objective-C هذه العلامة @ للدلالة على انها من النوع الجديد –تنطق هذه صوتياَ بقول "At" .

ترى ما التطوير الذي اوجدته Objective-C في النصوص وسلاسل الحروف عن سابقتها لغة C؟

اولاَ Objective-C تستخدم سلاسل حروف **Unicode** الشاملة بعكس سلاسل حروف **ASCII** المحدودة المستخدمه بلغة C .

ان سلاسل حروف **Unicode** الشاملة تمكنك من عرض اي سلسلة نصية بأي لغة حية تتخيلها من كالروسية، الصيني، العربية، بالاضافة الى اللاتينية **Roman alphabet** طبعاَ!

بالتأكيد يمكنك التصريح وتهئية محتوى متغير المؤشر بسلسلة نصوص ، دفعة واحدة [4] .

```
[4]
NSString *favoriteActress = @"Julia";
```

ان متغير المؤشر **favoriteActress** يشير الى عنوان حيز من الذاكرة حيث يوجد ذلك الكائن الذي يمثل ويحتفظ بالسلسة النصية **"Julia"** .

ما ان تنتهي من تهئية المتغير (**favoriteComputer** في مثالنا)، يمكنك عندها إسناد قيمة أخرى اليه، لاحظ انك لن تكون قادر على تغيير سلسلة النص ذاتها [3] ويعود السبب في ذلك كون هذه السلسلة النصية سليلة التصنيف **NSString** بسلسلة الوراثة . و سنتحدث عن ذلك خلال دقيقة .الان تابع القراءة. . .

```
[5]
#import <Foundation/Foundation.h>
 int main (int argc, const char *argv[])
{
  NSAutoreleasePool * pool =
[[NSAutoreleasePool alloc] init];

  NSString *x;
  x = @"iBook";  // [5.7]
  x = @"MacBook Pro Intel"; // Hey, I'm
just trying to make
                              // this book
look up to date!
   NSLog(x);
  [pool release];
  return 0;
}
```

عند تنفيذ هذا البرنامج، سيطبع لك السطر التالي :

```
MacBook Pro Intel
```

ان اي سلسلة حروف من سلالة التصنيف **NSString** تدعى **immutable** بمعنى انها غير قابلة للتغير والتعديل.

مالفائدة من وجود سلسلة حروف غير قابلة للتغيير ؟

بصراحة، يكون تعامل النظام مع سلاسل الحروف الغير قابلة للتعديل ايسر واسهل، مما يعنى ان برنامجك سيكون اسرع. في

الواقع سترى اثناء كتابتك للبرامج مستخدماَ Objective-C أنك غير مضطر لتعديل اي سلسلة حرفية في كثير من الاحيان.

وبالتأكيد هناك اوقات ستحتاج فيها ان تقوم بتعديل وتغيير سلسلة حرفية. لذا تم ايجاد تصنيف خاص يقوم بذلك، حيث سيمكنك من تعديل كائن النص بكل يسر وسهولة

هذا التصنيف معروف باسم NSMutableString .

وسوف نناقش طريقة التعامل معه لاحقاَ بهذا الفصل. ولكن دعنا اولاَ نؤكد لك ان سلاسل الحروف انما هى عبارة عن كائنات. ذلك يعني امكانية استقبالها للرسائل.

كمثال، يمكننا ارسال الرسالة **length** الى كائن سلسلة الحروف كما يلي بالمثال**[6]** .

```
[6]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool * pool =
[[NSAutoreleasePool alloc] init];
  int theLength;
  NSString * foo;
// Continued
```

```
  foo = @"Julia!";
  theLength  = [foo length];  // [6.10]
  NSLog(@"The length is %d.", theLength);

  [pool release];
  return 0;
}
```

عند تنفيذ البرنامج، سيطبع لك السطر التالي :

```
The length is 6.
```

يميل المبرمجين لاستخدام مفردتي **foo** و **bar** كأسماء للمتغيرات اثناء شرحهم للمواضيع المتعلقة بالبرمجة. في الواقع تلك المسميات سيئة جداَ، يعود السبب في ذلك لكونها غير ذات دلالة فلا معنى، انها مثل استخدامنا للمتغير **x**. ولكننا تعرضنا لها هنا حتى لاتشعر بالحيرة من امرك عندما تراها في نقاشات ومحاورات الانترنت.

في السطر **[6.10]** قمنا بارسال رسالة **length** الى الكائن **foo**، الرسالة تلك عبارة عن إجراء **method** مصرح عنه باسم **length** في تصنيف **NSString class** كما يلي :

```
- (unsigned int)length
```

انه إجراء يعود بقيمة عددية تمثل إحصاء لعدد الحروف الموجودة بسلسلة الحروف. ويمكنك إيضاَ تغيير الحروف الموجودة بالسلسلة الى حروف كبيرة **uppercase** .

ولتحقيق ذلك ارسل الرسالة المناسبة لكائن سلسلة الحروف ، نعني بذلك إجراء **uppercaseString** الذي يمكنك البحث عنه والاطلاع على طريقة تنفيذه من خلال قراءة وثائق المساعدة معتمداَ على نفسك هذه المرة (وتفحص الاجراءات التي يقدمها لك تصنيف **NSString**).

فور تلقيه للرسالة، يقوم كائن سلسلة الحروف بإنشاء وإرجاع كائن سلسلة حروف جديد يحوي نفس المحتوى ولكن تم تحويل الان كل حرف صغير **small letter** إلى ما يقابلة من الحروف الكبيرة **Capital Letter**.

```
[7]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool =
[[NSAutoreleasePool alloc] init];

//...Continued
```

```
  NSString *foo, *bar;
  foo = @"Julia!";
  bar  = [foo uppercaseString];
  NSLog(@"%@ is converted into %@.", foo,
bar);

  [pool release];
  return 0;
}
```

عند تنفيذ البرنامج، سيطبع لك السطر التالي :

```
Julia! is converted into JULIA!
```

في بعض الاحيان قد ترغب بتغيير محتوى موجود سابقاَ بدلاَ من ان تنشىء محتوى جديد. في مثل هذه الحالة عليك باستخدام الكائن الذي يأت من سلالة التصنيف **NSMutableString** حتى يتم عرض النص الجديد. فالتصنيف **NSMutableString** يوفر لك عدة إجراءات تمكّنك من تعديل محتوى سلسلة الحروف. كمثال، تجد أن الإجراء **appendString:** يقوم بإضافة سلسلة حروف الى نهاية سلسلة حروف الكائن المستقبل للرسالة.

```
[8]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool =
[[NSAutoreleasePool alloc] init];

  NSMutableString *foo;
     // [8.7]
  foo = [@"Julia!" mutableCopy];
     // [8.8]
  [foo appendString:@" I am happy"];
  NSLog(@"Here is the result: %@.",  foo);

  [pool release];
  return 0;
}
```

عند تنفيذ البرنامج، سيطبع لك السطر التالي :

```
Here is the result: Julia! I am happy.
```

ففي السطر [8.8] ، قام الاجراء **mutableCopy** (وهو من الاجراءات التي يقدمها لك تصنيف **NSString**) بإنشاء وإرجاع سلسلة حروف غير قابلة للتعديل بنفس محتوى سلسلة الحروف الكائن المستقبل. وذلك بعد الانتهاء من تنفيذ اوامر السطر [8.8] ، حيث كان المتغير **foo** مشيراَ الى سلسلة حروف غير قابلة للتعديل كانت تحتوى "**Julia!**".

في بداية فصلنا هذا اوضحنا الا يكون هناك اي تعامل مع كائنات **Objective-C** بشكل مباشر. وان اي عمليات تداول أو تعامل مع الكائنات يتم من خلال المؤشرات. واليك سبب ذلك ،فنحن قد استخدمنا متغير مؤشر بالسطر [8.7] . في الواقع عندما استخدمنا مفردة "كائن–**object**" إنما عنينا بذلك "مؤشر للكائن **pointer to an object**".

وطالما اننا نستخدم الكائنات بالمؤشرات دوماَ، فقد استخدمنا كلمة "كائن–**object**" إختصاراَ. ان حقيقة التعامل مع الكائنات من خلال المؤشرات له جانب آخر، ومن المهم جداَ استيعابه :

يمكن لعدة متغيرات ان تكون ممثلة –reference– لكائن من الكائنات في وقت واحد.

كمثال، بعد ان تم تنفيذ السطر [8.7]، اصبح متغير المؤشر foo مُمثلاَ لكائن يحتوي سلسلة الحروف المكونة لكلمة "Julia!"، هذه الامر يمكن تجسيدة على الصورة التالية :

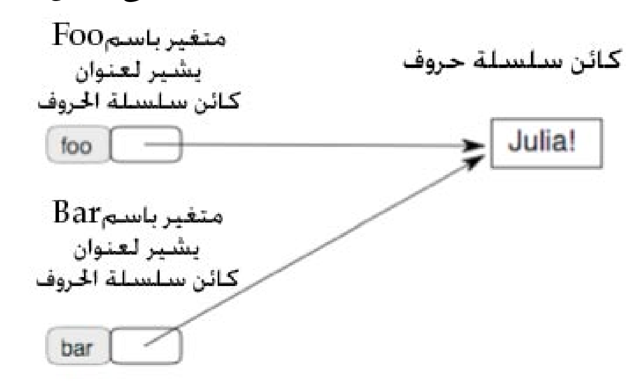


التعامل والتداول مع الكائنات يتم من خلال المؤشرات دوماً.

الان لنفترض اسنادنا قيمة المتغير foo الى المتغير bar كما يلي :

```
bar = foo;
```

نتيجة هذا الايعاز، ان كلا المتغيران سيشيران الى نفس الكائن.



يمكن لعدة متغيرات ان تشير الى نفس الكائن.

في هذه الحالة، عندما نرغب بارسال رسالة الى الكائن من خلال المؤشر foo (كأن نقول;[foo dosomething]) سيكون الاثر ذاتة كما لو قمنا باستخدام bar كأن نقول ;[bar dosomething])،

وهذا الامر موضح بهذا المثال :

```
[9]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool =
[[NSAutoreleasePool alloc] init];

  NSMutableString *foo = [@”Julia!”
mutableCopy];
  NSMutableString *bar = foo;

 NSLog(@”foo points to the string: %@.”,
foo);
  NSLog(@”bar points to the string: %@.”,
bar);
  NSLog(@”------------------------------”);

  [foo appendString:@” I am happy”];

  NSLog(@”foo points to the string: %@.”,
foo);
 NSLog(@”bar points to the string: %@.”,
bar);

  [pool release];
  return 0;
}
```

عند تنفيذ البرنامج، سيطبع لك ما يلي :

```
 foo points to the string: Julia!
 bar points to the string: Julia!
 ------------------------------
 foo points to the string: Julia! I am
happy
 bar points to the string: Julia! I am
happy
```

ان تكون لك القدرة على تمثيل (او الاشارة الى) نفس الكائن من عدة اماكن، ذلك يعد ميزة من اهم مميزات البرمجة بالكائنات. في الواقع لقد قمنا باستخدام هذه الخاصية في الفصول السابقة. ففي الفصل ٨ مثلاَ، قمنا بتمثيل الكائن **MAFoo** من خلال كائني الازرارين.

المصفوفات

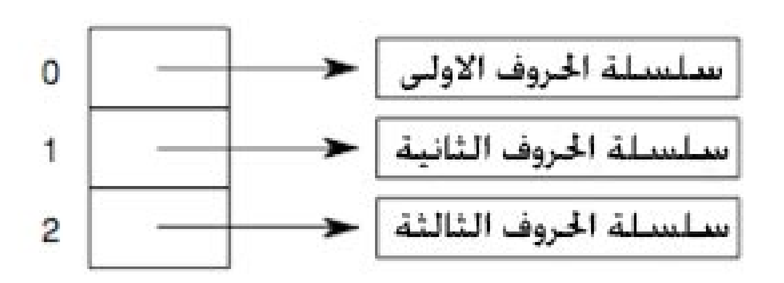
في بعض الاحيان يتطلب الامر منك ان تحتفظ بمجموعة متجانسة من البيانات. كأن تحتفظ بقائمة من سلاسل الحروف مثلاَ. وقد يكون الحل الغير عملي ان ننشيء متغيرات عديدة بحيث يحتفظ كل متغير منها بسلسلة حروف. ولكن الحل الافضل وبالتأكيد العملي، هو ان نستخدم المصفوفة -Array- .

فالمصفوفة عبارة عن قائمة مرتبة من الكائنات (أو مؤشرات للكائنات على وجه الدقة).

حيث يمكنك اضافة الكائنات بالمصفوفة أو حذف تلك الكائنات أو ان تستعلم المصفوفة عن ذلك الكائن برقمه التسلسلي (index الرقم الذي يحدد موقع ترتيب الكائن بقائمة المصفوفة)، يمكنك ايضاَ ان تستعلم المصفوفة عن العدد الاجمالي للكائنات التي تحتفظ بها.

وعندما نقوم باجراء عملية تعداد للعناصر فإننا دوماَ نبدأ عملية العد إبتداءَ بالرقم ١ ثم ٢ ثم ٣..الخ.

أما في المصفوفات على أية حال الامر مختلف فالعنصر الاول يبدأ بالرقم صفر (zero index–وهو الرقم التسلسلي الذي يقابل العنصر الاول بالقائمة) لذا سيكون الرقم التسلسلي للعنصر الثاني يساوي ١ ثم العنصر الثالث بمقابل تسلسلي يساوي ٢ ... وهكذا.



مثال : مصفوفة تحتوي ثلاثة (٣) سلاسل حرفية.

سنزودك بأمثلة للشفرة البرمجية لاحقاَ خلال هذا الفصل، لنمكنك من ملامسة الاثر الذي يحدثة التعداد ابتداءَ من الصفر. المصفوفات تأتيك كتصنيفان: التصنيف الاول NSArray والثاني NSMutableArray. كما هو الحال مع سلاسل الحروف strings.

فهناك نسخة غير قابلة للتغير والتعديل immutable وهناك نسخة مهمتها التغيير والتبديل mutable. في هذا الفصل، سنستهدف شرح النسخة القابلة للتعديل. وهناك طريقة واحدة لإنشاء المصفوفة، وهي تتم بالايعاز التالي:

```
[NSMutableArray array]
```

ان تنفيذ هذ الايعاز، سيعود علينا بمصفوفة فارغة. ولكن لحظة... هذا الايعاز يبدو غريباَ اليس كذلك؟ انه بالتأكيد غريب جداَ، ففي هذه الحالة قمنا باستخدام اسم NSMutableArray وحددناه ان يكون كمستقبل للرسالة.

الم يكن من المفروض ان نقوم بارسال رسائلنا إلى مستنسخ الكائن -instances- بدلاَ من التعامل مع تصنيف الكائن -class- ؟

في الحقيقة لقد تعلمنا شيء جديد: انها حقيقة من حقائق Objective-C حيث تتيح لنا ارسال الرسائل الى التصنيفات classes الاصلية (يعود سبب ذلك ان التصنيفات classes ذاتها عبارة عن كائنات، وبدورها هي ايضاً مستنسخات يطلق عليها meta-classes، ولكننا سنتوقف عن شرح هذه الفكرة لعدم مناسبتها لان تكون بكتاب يفترض ان يكون تقديمي للمبتدئين، وكذلك شرحها سيطول جدا).

من وثائق Cocoa، نجد ان الاجراءات التي يتم تنفيذها على

التصنيف ذاته تكون بادئتها بالرمز "+" بدلاً من الرمز "-" البادىء امام الاجراءات التي ترسل الى الكائنات. (انظر المثال [4.5] الذي ورد ذكره بالفصل ٨). وكمثال نرى ان الوثائق تصف إجراءات المصفوفة كما يلي:

array
+ (id)array
Creates and returns an empty array. This method is used by mutable subclasses of
NSArray.
See Also: + arrayWithObject:, + arrayWithObjects:

دعنا نطبق ما قرأناه سابقاً من خلال الشفرة البرمجية. فبرنامجنا التالي سيقوم بإنشاء مصفوفة فارغة، ليستودع بها ٣ سلاسل حرفية، ثم يقوم بطباعة العدد الممثل لمجموع العناصر بالمصفوفة.

```
[1]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool =
[[NSAutoreleasePool alloc] init];

//...Contined
```

```
  NSMutableArray *myArray = [NSMutableArray
array];

  [myArray addObject:@"first string"];
  [myArray addObject:@"second string"];
  [myArray addObject:@"third string"];

  int count = [myArray count];

  NSLog(@"There are %d elements in my
array", count);

  [pool release];
  return 0;
}
```

عند تنفيذ البرنامج، سيطبع لنا السطر التالي:

```
There are 3 elements in my array
```

برنامجنا التالي مشابه لما قبله الا ان هذا سوف يقوم بطباعة سلسلة الحروف الموجودة بالتسلسلي (صفر) داخل المصفوفة – اي العنصر الاول–. حتى نتمكن من الوصول لعنصر بالمصفوفة نستخدم الاجراء **objectAtIndex:** كما يلي**[2.13]**:

```
[2]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool =
[[NSAutoreleasePool alloc] init];

  NSMutableArray *myArray = [NSMutableArray
array];

   [myArray addObject:@"first string"];
   [myArray addObject:@"second string"];
   [myArray addObject:@"third string"];

  NSString *element = [myArray
objectAtIndex:0];  // [2.13]

  NSLog(@"The element at index 0 in the
array is: %@", element);

   [pool release];
  return 0;
}
```

عند تنفيذ البرنامج سيطبع السطر التالي :

```
The element at index 0 in the array is:
first string
```

سيتطلب منك دائما ان تسبر المصفوفة حتى تتمكن من عمل اي إجراءات على اي عناصر مستودعة بها. وللقيام بذلك يمكنك استخدام حلقات التكرار كما سنرى في البرنامج التالي الذي سوف يقوم بطباعة محتوى كل ما سيقابلة من عناصر مستودعة داخل المصفوفة.

```
[3]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool =
[[NSAutoreleasePool alloc] init];

  NSMutableArray *myArray = [NSMutableArray
array];
[myArray addObject:@"first string"];
  [myArray addObject:@"second string"];
  [myArray addObject:@"third string"];

  int i;
  int count;
  for (i = 0, count = [myArray count]; i <
count; i = i + 1)

//... continued
```

```
    {
     NSString *element = [myArray
objectAtIndex:i];
     NSLog(@"The element at index %d in the
array is: %@", i, element);
   }

   [pool release];
   return 0;
}
```

عند تنفيذ البرنامج سيقوم بطباعة مايلي على الشاشة:

```
The element at index 0 in the array is:
first string
The element at index 1 in the array is:
second string
The element at index 2 in the array is:
third string
```

لاحظ للمصفوفات القابلية ان تحتوي اي كائنات اخرى فهي ليست محصورة بتخزين سلاسل الحروف فقط.

يزودك تصنيفي **NSMutableArray** و **NSArray** بعدة إجراءات، وننصحك ان تطالع ما تحتوية وثائق المساعدة لهذين التصنيفين حتى تتمكن من التعرف اكثر على فوائد ومميزات المصفوفات.

وسنقوم بانهاء هذا الفصل من خلال حديثنا عن الاجراء الذي يمّكنك من تبديل عنصر معروف رقمه التسلسلي بعنصر آخر داخل المصفوفة. اسم هذا الإجراء هو **:replaceObjectAtIndex withObject:** فحتى هذه اللحظة قمنا بالتعامل مع الاجراءات التي تطلبت وجود معامل واحد كحد أعلى. وهذا الاجراء هنا مختلف، ولذلك نحن نتفحصة الان، فهذا الاجراء يتطلب وجود معاملان. ويمكنك ان تؤكد ذلك حيث يوجد عدد ٢ من النقاط العمودية ":".

تسمح **Objective-C** للإجراءات بأن تحتوى اي عدد من المعاملات **-arguments-**. وهنا نرى كيف يمكننا استخدام هذا الاجراء.

```
[4]
[myArray replaceObjectAtIndex:1
withObject:@"Hello"];
```

بعد تنفيذ هذا الاجراء، سيكون العنصر الثاني ذي الرقم التسلسلي ١ محتوياَ لسلسلة الحروف **@"Hello"** . بكل تأكيد يجب تنفيذ هذا الاجراء بوجود رقم تسلسلي موجود. بمعنى يجب ان يكون هناك عنصر مخزن بالرقم التسلسلي المذكور، حتى يتمكن الاجراء من تنفيذ عملية الاستبدال بالكائن قيد البدل.

كما ترى من المثال، ان اسماء الإجراءات بـ **Objective-C** مثل الجمل التي تتطلب منك ان تملء فراغاتها. (الفراغ المطلوب ملئه

يأتي بعد النقطتين العموديتين "إجراء معامل١ : ..... معامل٢ : ..... "). ومتى ما وضعت الإجراء قيد التنفيذ وجب عليك ملء تلك الفراغات بقيم حقيقية، حتى تتمكن من إنشاء جملة ذات معنى.

هذا الاسلوب المتبع للتسمية وتفعيل صياغة الإجراءات قد واضح جداً بـ Objective-C بسبب ما ورثتة من سلفها لغة **Smalltalk** البرمجية. مما اكسب Objective-C قوتها الحالية بحيث اصبحت شفرتها البرمجية مُصاغة بشكل يجعلها مقرؤة بكل وضوح وتلقائية.

عندما تقوم بتأليف إجراءاتك الخاصة، احرص على ان تكون مسمياتها مصاغة بشكل يجعلها تُقرأ كافعال بجملة. هذا الاسلوب يعين من تسهيل قراءة الشفرة البرمجية، وهو امر مهم لان تكون برامجك قابلة للفهم مما يسهل من عملية تصيينها.

إدارة الذاكرة

قدمت اعتذاري باكثر من فصل سابق، حول تفسير بعض الايعازات التي سوف نتناولها هنا. فهذه الايعازات تتعامل مع الذاكرة .فكما ترى إن برنامجك ليس البرنامج الوحيد بجهاز الماكنتوش، حيث تكون الذاكرة RAM محدودة كالعملة الصعبة. لذا في حال لم يعد برنامجك بحاجة لذلك الجزء المحجوز من الذاكرة، يجب عليه عندها إعادة ذلك الجزء إلى النظام.

لقد احسنت والدتك تربيتك عندما اخبرتك بأن تكون متأدباً وان تتعايش مع باقي أفراد مجتمعك، لقد كانت تعلمك بطريقة ما كيف يكون البرنامج!

وحتى لو كان برنامجك هو الوحيد قيد التشغيل، فإن عدم فك الحجز عن الذاكرة وإستهلاك المزيد يؤدي الى حكر برنامجك تدريجياً بركن يصعب الخروج منه، وسيكون أداء حاسبك بطيء جداً لدرجة الحبو.

عندما يقوم برنامجك بإنشاء كائن جديد، سيحتل الاخير جزء من مساحة الذاكرة وسيكون من واجبك إخلاء وفك حجز ذلك الجزء من الذاكرة متى ما انعدمت حاجتك لوجود ذلك الكائن. ان

انعدام حاجتك للكائن تحتم عليك إفناءة / تدميره وإخلاء المساحة التي كان يحتجزها .على كل حال، تقرير مصير الكائن من خلال مدى حاجتك قد لا يكون بالامر اليسير.

كمثال، واثناء تشغيل البرنامج، قد يكون كائنك هذا مستخدماً من قبل عدة كائنات تشير اليه مرجعياً، عندها يجدر بك الا تقوم بتدميره لاحتمال ان يقوم احد تلك الكائنات الاخرى بإستخدامه.

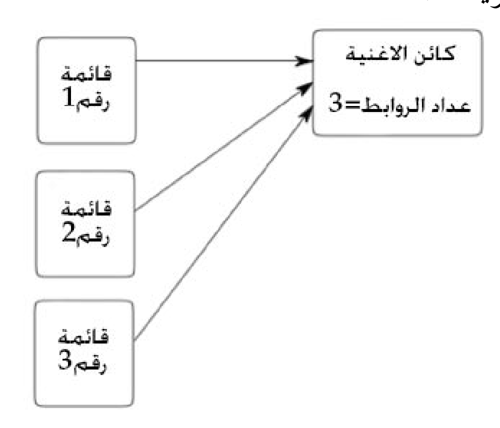
(ان مجرد محاولة استخدام كائن تم تدميره تؤدي الى توقف برنامجك عن العمل crash أو يتصرف بطريقة غير المقررة)

وللمساعدة في تقرير عملية إفناء الكائن عند انعدام الحاجة إليه، تقدم لك Cocoa مبدأ ربط الكائن بالعداد، وهو يمثل ما يطلق عليه "عداد الروابط **retain count**" الخاص بكل كائن.

ففي برنامجك متى ما عينت اشارة مرجعية **-reference-** لاحد الكائنات، وجب عليك عندها اعلام ذلك الكائن، حيث تقوم بزيادة عداد روابطه بقيمة مقدارها ١ . ومتى ما قمت بإزالة تلك الاشارة المرجعية، وجب عليك القيام بإنقاص عداد روابطه بقيمة مقدارها ١ .

متى ما اصبحت قيمة عداد الروابط مساوية لصفر، يعلم الكائن عندها انه لا يوجد احد مرتبط به وانه يمكنه عندها تدمير ذاته، مخلصاً بذلك الحيز الذي كان يحتله من الذاكرة.

كمثال، لنفرض ان برنامجك عبارة عن مشغل موسيقي ولديك كائنات تمثل "الاغنيات" وأخرى تمثل "قوائم اسماء الاغنيات" . لنفرض ان كائن "اغنية" مرتبط بإشارة مرجعية من قبل ٣ كائنات "قوائم أسماء الاغنيات" . عندها ستكون قيمة عداد روابط الكائن "اغنية" مساوية لـ٣ .



يستطيع الكائن معرفة عدد الاشارات اليه.

لقد اصبح للكائن القدرة على معرفة عدد المرات التي تمت الاشارة اليه، والفضل في ذلك يعود الى عداد الروابط. وللقيام بزيادة

عداد الروابط لاحد الكائنات، كل ما علينا عمله هو ارسال رسالة **retain** لذلك الكائن. اما في حال رغبتنا بإنقاص عداد الروابط عندها نرسل له رسالة **release** .

تقدم لك **Cocoa** آلية عمل آخرى تدعى "**autorelease pool**" وهي تمكّنك من ارسال رسالة **release** متأخرة لذلك الكائن. – **delayed message** بمعنى انها ليست قيد الارسال حالاً بل بعد تخضع لفترة متأخرة من الزمن. ولاستخدام هذه الآلية، كل ما عليك عمله هو تسجيل ذلك الكائن بقائمة ما يسمى "**autorelease pool**"، من خلال ارسال رسالة **autorelease** . وستقوم عندها آلية "**autorelease pool**" بالتكفل بعملية ارسال رسالة **release** المتأخرة إلى ذلك الكائن المسجل معها.

إذاً الايعازات التي تتكفل بها "**autorelease pool**" وقد رأيناها في الشفرة البرمجية بالأمثلة السابقة، هي ايعازات موجهة لنظام التشغيل حتى يتكفل بتهيئة آلية عمل "**autorelease pool**".

ان عملية ادارة الذاكرة المستخدمة من قبل **Cocoa** والتي تم عرضها بهذا الفصل تُعرف عادة بمسمى "**reference counting**". وسوف تجد شرحاً وافيا حول نظام **Cocoa** لادارة الذاكرة بكتب و مواضيع متقدمة (انظر الفصل ١٥).

افصح بعض مهندسي أپل عن توجهها لتطوير نموذج آلية جديدة تعرف باسم "**automatic garbage collection**". حيث ستكون هذه الالية اكثر قوة من الوسائل الحالية، وذات استخدام اسهل وبنسبة اخطاء اقل وقت كتابة هذا الكتاب.

على كل حال لا توجد ضمانه عما اذا كانت هذه التكنولوجيا مكتملة، أو متى ستقوم أپل بتزويدها في **Cocoa**.

مصادر المعلومات

ان الهدف الاسمى لهذا الكتاب هو تعليمك اسس لغة Objective-C من خلال بيئة Xcode. فاذا صدف ان قرأت هذا الكتاب مرتين، وقمت بتجريب الامثلة واضفت تجاربك عليها، عندها تأكد انك جاهز لان تتعلم كيفية عمل برنامجك الفتاك، الذي طالما رغبت بكتابته.

هذا الكتاب يزودك بمفاتيح المعرفة الاولية، و الكافية لتتعمق المشاكل البرمجية بشكل اسرع. وبما انك لا تزال معنا حتى هذا الفصل، فانك حتماً جاهز لان تغزو المصادر المعلوماتية الاخرى، وما سنذكره هنا سيلفت انتباهك.

نصيحة مهمة قبل ان تبدأ بكتابة شفرتك البرمجية: لاتبدأ بإندفاع، أبداَ ! بل تروى قليلاَ ! وتفحص هياكل العمل frameworks المتوفرة...

ذلك ان أپل، غالباَ ما قامت عنك بالعمل، أو زودتك بتصنيفات تتطلب القليل من التعديل لتحقق ما تريد. كذلك من الممكن ان يكون احدهم قد قام بإنجاز ما تحتاج تصميمه واتاح شفرته المصدرية -open source- للجمهور. لذا وفر على نفسك العناء

من خلال مطالعتك لوثائق المساعدة والبحث بالانترنت. واول مصدر يجدر بك زيارته هو موقع مطوري أپل بالرابط

```
Apple's developer site at: http://www.
apple.com/developer
```

نحن ننصحك بشدة ان بإرتياد وتبويب -bookmark- العناوين التالية:

```
http://osx.hyperjeff.net/reference/
CocoaArticles.php
http://www.cocoadev.com
http://www.cocoabuilder.com
http://www.stepwise.com
```

المواقع الموضحة بعالية تحتوي روابط متنوعة ومتعددة لمواقع أخرى ذات مصادر معلومات متشعبة وشاملة. يجدر بك تسجيل عضويتك بالقائمة البريدية الخاصة بموقع **cocoa-dev** على الربط

```
http://lists.apple.com/mailman/listinfo/
cocoa-dev
```

ذلك المكان الذي يمكنك طرح اسئلتك فيه بكل حرية. حيث سيقوم العديد من الاعضاء بتقديم يد العون بكل حفاوة وترحيب. ولكن مقابل هذه الحفاوة يتطلب منك القيام أولاَ بتفحص ما يحتوية الارشيف من اجوبة قد تكون شافية لاستفسارك أو حلاَ لمشكلتك، ان رابط الارشيف هو

```
http://www.cocoabuilder.com
```

وللمعلومات والنصائح حول اتيكيت فن التخاطب وطرح الاسئلة على اعضاء القائمة البريدية، إقراء **"How To Ask Questions The Smart Way"** التي ستجدها بالرابط

```
http://www.catb.org/~esr/faqs/smart-
questions.html
```

هناك عدة كتب قيّمة حول التطوير باستخدام **Cocoa**.و نبدأ سردنا بكتاب **Programming in Objective-C, by Stephen Kochan** وهو موجه للمبتدئين. وبجعبتنا بعض الكتب التي تفترض اكتسابك لمعرفة برمجية كالتي استفدنا منها بهذا الكتاب. لذا ننصحك بكتاب **Cocoa Programming for Mac OS X by Aaron Hillegass** والمؤلف صاحب مؤسسة **Big Nerd Ranch** التعليمية حيث يعطي دورات في البرمجة تؤهلك للعمل بمهنة "مبرمج محترف". كذلك ننصحك بكتاب **Cocoa with Objective-C by James Duncan Davidson and Apple** وهو من اصدارات **O'Reilly**. وهناك كتاب **Cocoa Programming by Scott Anguish, Erick M.Buck, Donald A.Yacktman** وهو من إصدارات **SAMS** والكتاب دسم جداَ بكبر القاموس.

كلمة أخيره بهدف التنبيه.

وانت على وشك ان تبرمج برامجك على الماكنتوش. وقبل ان تقوم بإطلاق برنامجك للجمهور، لا تكن حريصاَ على ان يكون برنامجك خالياَ من الاخطاء البرمجية فقط، بل احرص على ان يظهر برنامجك بالشكل اللائق وان يحقق المواصفات التي وضعتها أپل لتصميم الواجهات الرسومية **Apple human interface guidelines** والتي ستجد مادتها القيّمة بالرابط

```
http://developer.apple.com/documentation/
UserExperience/Conceptual/OSXHIGuidelines/
index.html
```

فما ان تنتهي من ذلك، إدفع ببرنامجك إلى الجميع! ان التعليقات التي ستتلقاها من نقد وتقييم ستكون معيناَ لك لتطوير وتحديث برنامج قيّم مادمت تحرص على ذلك بتركيز.

نتمنى ان تكون قد استمعت معنا بقراءة هذا الكتاب، وان ذلك سيكون دافعاَ لك لسبر اغوار البرمجة بإستخدام **Xcode**، والرجاء الرجاء لا تنسى ما ورد بالفصل رقم ٠

Bert, Alex, Philippe.

مسرد المصطلحات

ampersands: &&. ٤٣, ٩٩

اسم للرمز & المستخدم لإجراء عمليات المقارنة المنطقية والتي تفترض استيفاء كافة الشروط.

Apple ١, ٩٨

شركة أپل المطورة لنظام ماكنتوش والتكنولوجيا المصاحبة.

Application heading ٥٤, ٨١

اللازمة البادئة للتصريح عن المصادر والمتغيرات والمحتويات المطلوبة لإجراء تجميع الشفرات البرمجية بالملفات الرقمية.

garbage collection ٩٩

بروتوكول بإتفاقية معينة لتوفير وتحرير مصادر الذاكرة.

Cocoa Application ٥٤, ٥٥

برنامج مبني من خلال ما توفره تكنولوجيا و مكتبات Cocoa البرمجية.

Code snippest ٢, ١٦

جزء بسيط من الشفرة البرمجية بغرض العرض والايضاح.

headers ٣٨, ٥٢, ٥٤, ٥٨, ٦٣, ٩٢, ٩٩

انظر Application heading

human interface guidelines ١٠٣

عدد من الضوابط المحددة لتوضيح التصميم الامثل للواجهة الرسومية للبرنامج ليصلح للتعامل الانساني.

IB / Interface Builder ??

مصمم واجهة البرامج الرسومية. للتمكن من وضع وضبط مواقع العناصر، وتوصيلاتها.

object-oriented programming ٧٣

اسلوب برمجي لبناء البرامج من خلال استخدام الكائنات، وما يتبع ذلك من ضوابط وقواعد لغوية تميز هذه اللغة.

Objective-C ١, ٣

تفّرع من لغة C الام يهدف للبرمجة من خلال توظيف الكائنات.

OR ٤٣
الرمز (أو) المستخدم لإجراء عمليات المقارنة المنطقية .

string ٧٠, ٨٦, ٩٩
سلسلة من الحروف المكونة للنّصوص المقروؤة .

unsigned int ١٢, ٧٣, ٩٨
اي رقم صحيح غير سالب .

إجراء ٥٢
مجموعة الاوامر والايعازات التي تحدد مهام عمل البرنامج .

إفناء / تدمير الكائن ٩٨
عملية تحرير جيز الذاكرة المحجوز للإحتفاظ بقيم عنصر كائن برمجي إنتهت الحاجة منه .

إقحام implement ٧٦
دمج وتضمين توصيف الإجراءات التي يجب إتباعها عند تنفيذ الكائن للأوامر .

إيعاز ٨
كلمات ومفردات إجرائية محجوزة لعمل مهمة معينة .

اخطاء البرمجة ٤, ٨, ٩, ٣٦
إدخالات غير صحيحة لمفردات اللغة، او عمليات حسابية غير دقيقة .

ارتباط ٢
مؤشرات لروابط مؤدية لمزيد من المعلومات ذات العلاقة .

ازرار Button ٥٠
تشبيه مجازي للمفاتيح التي يتعامل معها البشر في العالم الخارجي .

اشارة مرجعية Referenc ٩٨
عنصر مؤشر كمرجع لكائن او عنصر برمجي .

اطار Frame ٣٣
تشبية مجازي لحاوية تجمع عدد من العناصر

تدوير شفرة البرمجة. code reuse ٢٣
إمكانية إعادة إستخدام الإجراءات لعمل مهامها مع عدد من المتعاملين .

اعداد كاملة ١٠
الاعداد الرياضية الصحيحة والخالية من الكسور .

اعداد كسرية fractional ١٠,٩٢
الارقام الحقيقية ذات الفاصلة العشرة مثل العدد ٥٫٥ او العدد ٢٫٧٥

اكواد تشغيلية ٣١
كلمات محجوزة للغة لتنفيذ مهام مهينة – انظر الإجراءات

الاجراءات المتلازمة Accessors ٧٢
تؤام من الإجراءات تنحصر مهمة الاول بإسناد قيمة بينما الثاني إسترجاع هذه القيمة .

الاقواس المعقوفة {} ٢٣, ٣٤
كتل محددة للشروط والمعطيات التي تتطلبها بعض الإجراءات والمفردات المحجوزة باللغة .

الايعاز الشرطي المعشش ٤٣
إستفسارات حول قيم منطقية بحيث يكون كل استفسار مضمن ضمن استفسار آخر

البايت ١١
أصغر وحدة يستخدمها الحاسب لتخزين المعلومات .

البرمجة ٢

عدد من التوجيهات والاوامر التي يقوم الحاسب بتنفيذها.

البروتوكول ٧٦

إجراءات ذات صيغة وترتيبات عمل موحدة

الترتيب الافتراضي ٣٣

النسق المعد او القيمة المسندة إفتراضياً وقت إنشاء العنصر أو الكائن.

التصريح عن المتغير ١١

عقد إتفاق لحجز حيز من الذاكرة يدعى بالتصريح عن نوع واسم المتغير.

التصفح ٧٧

إستعراض المعلومات وفق مظهر وترتيب معين.

التوصيلات ٦٣

حجز متغيرات للعناصر من خلال التطبيق IB

التوقف المؤقت Braek Point ٣٨

حالة من التوقف المؤقت لتفحص معطيات العناصر البرمجية بالبرنامج قيد التفحص.

الخلية ٧١

تشبيه مجازي لعدد من العناصر المكررة عمودياَ أو أفقياَ او الاثنان معاً.

الدالات الوظيفية ١٧

مجموعة من الإجراءات تؤدي وظيفة معينة.

الدوال / الدالات البرمجية function ١٨

مجموعة من الإجراءات تؤدي وظيفة معينة مع إرجاع قيمة معينة.

الساطرة السفلية "ـ" Underscore ٩

رمز حرفي من الحروف المتوفرة بلوحة المفاتيح

السحب والافلات ٥٦

اسلوب تعامل فريد من مواصفات انظمة تشغيل أبل.

الشرط المنطقي ٤١

عملية إستيفاء لشرط تنحصر عمليات القيم فيه بصحيح او غير صحيح.

الشفرة البرمجية Code ٢

رموز ومفردات اللغة البرمجية حيث تكون عبارة عن كتل تعرف بالشفرة البرمجية المستخدمة لبناء البرنامج.

الفأرة ٥٠

جهاز يزّود المستخدم بمؤشر على شاشة الحاسب، كتشبية مجازي ليد المستخدم.

القائمة العليا Manu Bar ٥٨

مجموعة الإجراءات المتاحة للمستخدم حسب البرنامج قيد التشغيل وتوجد بالمساحة العليا من الشاشة.

الماكنتوش Macintosh ٣

نظام تشغيل بتكنولوجيا متطورة يعتمد مكتبات رسومية وكائنات برمجية قابلة لإعادة الاستخدام بضوابط معينة.

المتغيرات Variables ٨

حاويات تخزن قيم بانواع معينة، تحتل جزء من ذاكرة الحاسب الالي.

المجّمع compiler ٨

برنامج يمثل المرحلة التي تلي تفسير الشفرة البرمجية وجلب المصادر اللازمة لتحويلها الى لغة الالة.

المسائل البرمجية ٧

عملية تحويل مشكلات حسابية ومتطلبات

عمل ومهام العالم الخارجي الى صيغ رياضية ومعادلات قابلة للحل من خلال الحاسب.

المصفوفات ٩١
نمط من المتغيرات التي تخزن قيم / كائنات متجانسة.

المُعامِلات الحسابية ١٢
العلميات الحسابية الرياضية التي يتم تطبيقها على الاعداد.

المنصة Dock ٥١
شريط يحتفظ بمؤشرات للبرامج التي يستخدمها او يعتمدها المستخدم.

النافذ الرئيسة ٥٦
النافذة الرئيسة التي تمّثل كافة تعاملات المستخدم مع البرنامج قيد التشغيل.

النطاق المحلي للدالة variable scope ٢٣
عملية حجب وحماية اسماء التغيرات ومحتوياتها داخل نطاق الدالة دون التعارض مع تشابة الاسماء في حال وجد خارج نطاقها.

النموذج الاب superclass ٥٨
توصيف للكائن البرمجي بحيث يستقي منه الكائن الوريث او الكائنات الوريثة ما يحتويه الاب من مواصفات وإجراءات.

النموذج التصنيفي Class Template ٥٢
موصفات للتصنيف تستخدم كنماذج هيكلية لبناء ما يلي من كائنات.

الهوامش ٣٨
عملية ترتيب وتنسيق نصوص الشفرة البرمجية بمساحات فارغة لسهولة القراءة وتعيين التعشيش الخاص بكل دالة.

الواجهة الرسومية للنظام ٤٠
التشبيهات المجازية للعناصر بالعالم الخارجي، داخل شاشة الحاسب من إزرار ونوافذ وكائنات رسومية.

دالات تعيد قيمة return a value ٢١

انواع البيانات ٢٠ ,٨٣
للبيانات انواع منها العدد ومنها النص ومنها الكائن، حتى يتعامل الحاسب معها يجب التصريح عنها لتوفير الحجز المناسب من الذاكرة المتاحة.

برمجيات ٤
برامج تحقق نفعية معينة دون ان تكون من ضمن البرامج الكبيرة.

برنامج Safari ٥٠
برنامج أپل لتصفح مواقع الانترنت.

برنامج Xcode ١
واجهة إنشاء البرامج وتجميع الشفرات البرمجية من خلال عدة لغات مثل coca, carbon,java

تقصي الاخطاء البرمجية debugger ٣٩
برنامج متخصص لتفحص اخطاء ومعطيات وسير عمل البرنامج قيد البناء.

بُنية الاجراء ٦٩
كتل الشفرة البرمجية التي توصف سير عمل هذا الاجراء.

بنية الدالة ٢٣

انظر بنية الإجراء

بنية غير مهيكلة unstructured ١٩
برنامج مكون من عدد من المكونات الغير مرتبة بشكل منطقي.

ترويسات headers ٣٨, ٥٤
انظر

ترويسة التطبيقات
نماذج مبيته من المصادر والكتل البرمجية وهي جاهزة للإستخدام الفوري.

تصريحات ٧١
عقد اتفاق مع الحاسب لحجز حيز من الذاكرة لمتغير أو كائن مصّرح عنه.

تصنيف Class ٥١, ٥٧
كائن برمجي من سلالة معينة بعدد من التوصيفات والاجراءات والوظائف المفيدة.

تصنيف اعلى Root Class/Super Class ٥٧
الاب المؤّرث في سلسلة الوراثة

تصنيف متحدر sub-Class ٥٨
الابن المتحدر من سلسلة الوراثة.

تطبيقات ١
محموعة البرامج التي تؤدي ادوار ومهام برمجية

تعاودية ٥٨
إستدعاء الإجراء لذاته ضمن إجراءاته الداخلية.

تعليق مقتضب ٥٦
تفسير مختصر دون التعمق لشرح الفكرة.

تفسير ٢٢
عملية تقييم مفردات الاجراءات البرمجية فور تلقيها، وهي عملية بطيئة نسبياً.

تكييف override ٥٨
تغير طفيف في أداء عمل مصنف او إجراء بالبرنامج

حقل نص text field ٥٢
تشبية رسومي مجازي للحقل المتاح لإستقبال ادخالات المستخدم من النصوص.

حلقة do {} while () ٤٧
كتلة محتوية لعدد من الإجراءات، والتي تنفذ وفق عدد معين من المرات بناء على شرط منطقي محدد من قبل المبرمج.

حلقة التكرار for-loop ٤٦, ٤٧, ٩٤
كتلة محتوية لعدد من الإجراءات، والتي تنفذ وفق عدد معين من المرات.

خصائص proprity ٥٦
قيم مكنونات وخصائص العنصر البرمجي أو الكائن.

خطوات الزيادة ”steps“ ٤٦

دالة main() ١٨
الدالة الرئيسة بشفرة البرمجة عند تشغيل اي برنامج.

سطر الاوامر Command Line ٣٣

سلالة التصنيف ٥٨
نوع السلالة التي يرث منها الكائن صفاته.

سلسلة الحروف string ٢٦
سلسلة الحروف المكونة للنصوص، حيث يتم الاشارة لمحتوياتها من حروف وكلمات.

سلسلة التوارث / الوراثة ٥٧, ٨٣

موقع تصنيف الكائن من السلالة التي يرث منها صفاته.

شريط الأدوات Tool Bar ٥٥
تشبيه مجازي لشريط محتوي لأزرار تؤدي عدد من الإجراءات.

عداد الروابط ٩٨
تكنيك برمجي مستخدم للحفاظ على مصادر الذاكرة وإفناء الكائنات الغير مستخدمة.

علامة المربع # ٣٤
رمز المربع المستخدم لجلب مصادر وكواد برمجية ببادئة الشفرة البرمجية.

فاصلة منقوطة ";" ٨
رمز الفاصلة المنقوطة، حيث تستخدم للإستدلال على إنتهاء السطر البرمجي.

قائمة منسدلة pop-up menu ٣٩, ٥٦
نافذة مبيته لعذد من الإجاراءات والاوامر، تعرض محتوياها وقت الضغط عليها.

قيمة Value ١١
الموّصف العددي او النصّي او الكائن المستخدم لإجراء عمليات مفارنة وكميات.

قيمة افتراضية Default Value ٧٧
قيمة معينة تستخدم وقت إنشاء المتغير أو الكائن.

كائنات برمجية ٥٠
موصّفات برمجية لمجموعة من الإجراءات والوظائف.

كلمة محجوزة ١٨
لكل لغة برمجية كلماتها المحجوزة للدلالة على تنفيذ إجراءات بعينها.

لغة C ٥٧, ٧٩
لغة البرمجة البنيوية من اساس **Ansi** والتي تخاطب الالة من خلال عدد محدود من المفردات والكتل البرمجية المفهومة للبشر.

لويحة Tab ٦١
معّرف رسومي كتشبية مجازي لتبويب معين من البيانات والمعطيات.

مجلد الادوات Utilities Folder ٤, ٩٨
مجلد ادوات البريمجات المساندة بالنظام.

مجلد التطبيقات Applications ٤
مجلد البرامج بالنظام.

مجموعة متجانسة ٩١
مجموعة من العناصر التي تشترك بذات التركيب البنيوي أو الإجراءات والصفات مع اختلاف القيم المسندة.

مراقب النشاطات Activity Monitor ٤
بريمج مساندة لمراقبة اداء النظام وحصر للبرامج قيد التشغيل مع مدى المصادر المستهلكة

مستنسخ التصنيف class instance ٥١
مؤشر خفيف كنسخة مستقلة لّموصف الكائن البرمجي النموذج.

مستوى جذري واحد Flat Level ٣٣
تساوي ترتيب كافة العناصر على مسطح او متسوى واحد.

معامل الدالة Argument ٢٠, ٥٣
القيمة/القيم أو الكائن/الكائنات اللازمة إدخاله ضمن تنفيذ تلك الدالة.

معشش Nesting ١٩
تعشيش، بمعنى تضمين عمليات إجوائية، داخل عمليات إجرائيةأخرى.

ملامح Attributes ٥٦
السمات التي تحدد هيئة العناصر البرمجية بصرياً.

منفذ خرج Output Port ٦١
مصدر لتلقي مخرجات عمليات الحاسب الالي.

موقع أپل الالكتروني ١
موقع الشركة الام حيث يتم تزويدك بالمصادر والتعليمات المتعلقة بالنظام واساليب برمجته.

نافذة ٥٠
عنصر برمجي يؤدي دور احتواء وعرض عناصر برمجية كتشبيه مجازي للحاوية.

نافذة التوثيق Run Log window ٣٦
نافذة لعرض مخرجات عناصر البرنامج.

نطاق الدالة ٢٠, ٢٣
محدد لاسماء المتغيرات التي تنحصر داخل نطاق الدالة بعينها.

نظام Mac OS X ١
نظام تشغيل حديث مكون من عدة مكونات لانظمة تشغيل اهمها نظام BSD، وتكنولوجيا ابل من مكتبات cocoa و car-bon ومدير المكونات الرسومية quartz.

نوع البيانات العائد ٢٠
انظر نوع Type - حيث تشترط بعض الدوال إسترجاع قيمة من نوع معين.

هيكلة structure ١٩
نظام بنيوي مرتب لتكوين بنية البرنامج منطقياَ حتى يسهل تطويرة وزيادة قدراته.

واجهة التطبيقات الرسومية ١
واجهة تعامل رسومية مع الحاسب الالي، تستند في إجراءاتها ومعاملاتها علىالتعامل التشبيهي للعناصر من ازرار ونوافذ، هذه الواجهة تستخدم النصوص والرسومات.

وحدة طرفية Terminal ٣٢
واجهة تعامل مع الحاسب الالي، تستند على إجراءات ومعاملات تُدخل يدوياً، هذه الواجهة تستخدم النصوص فقط.

وندوز Windows ٣
نظام تشغيل رسومي معّد من قبل مايكروسوفت

ملاحظات المترجم

لقد حرصت كل الحرص ان اقوم بنقل المعلومات الواردة بهذا الكتاب BECOME AN XCODER بكل دقة وامانة مع مراعاة قواعد اللغة العربية، ونظراَ لاختلاف طبيعة بعض المسميات التي لم يأت فيها الاتفاق عندما يختص الامر بالبرمجة الحديثة كإستخدام الكائنات حيث يشير البعض اليها بالبرمجة غرضية التوجة، وهناك فريق يرى انها البرمجة بالأشياء لذا اعتمدت الإشارة الى كل ما يتعلق بهذا القطاع الجديد من البرمجة بالكائنات وما يندرج تحتها من مصنفات ومستنسخات، ومؤشرات pointers نائبة، لإنابتها عن الكائن الحقيقي أثناء التعامل والتداول. لذا قد يظهر للسطح ازدواجية في استخدام بعض المفردات التقنية، والتي قد يعتاد من هم بحقل البرمجة عليها من تسميات معّربة ككلمة Class حيث اعتمدتها بكلمة "تصنيف" في هذا الكتاب رغم ان بعض المراجع تراها "كطبقة" . كذلك اعتمدت كلمة إجراء ودالة للإشارة الى كل من Method,Action وfunction، و لم امتنع عن سرد هذه المصطلحات مع مترادفاتها اللاتينية، وقد اشير اليها بطريقة او أخرى، وفي حال اختلال المعنى يكون الحل بمسرد المصطلحات.

حول هذا العمل

البرمجة من وجهة نظري نوع من انواع الهوايات التي امضيت معها فترة تتجاوز العشرة سنوات وانا اراقب تطورها وتحسنها دون مشاركة فعلية مني بتطوير ما يمكن ان يكون برامج عربية ذات مقاييس موحدة، وقيمة منفعية جمة لمن يستخدم تلك البرامج.

بالصدفة اثناء تجولي باحد المواقع، وجدت هذا الكتاب القيم، وهو دعوة لنشر ثقافة ووعي جيل جديد من المبرمجين، وقد رأيت انها فرصة سانحة لاثبات مدى فهمي من خلال ترجمة ما يحتوية الكتاب، مؤلفيه الاصليين يشجعون الجميع على قراءته وتداوله وهو بالفعل ذي قيمة عالية.

لقد اعتمدت في تنسيق هذا الكتاب نمط موحد من الخطوط وهو خط البيان Al Bayan المزود بالنظام وهو مستخدم للنصوص الداخلية بمقاس مقداره ١٨ نقطة وهو ذاته المستخدم للتبويب والترويسات العلوية (١٣ نقطة) والعناوين الرئيسة (٣٦ نقطة)، اما العناوين الفرعية (**نمط عريض**) فهي ذاتها المستخدمة لبُنية النص اما التنويهات فقد اعتمدتها بمقاس ١٢ نقطة بنسبة ٥٠٪ من الاسود بالكتاب، اما الخط اللاتيني المستخدم هو Palatino (T1) مقاس ١٢ نقطة كبنط انسب، اما لنمط خطوط الشفرة البرمجية فقد رأيت ان الخط `Courier New` مقاس ١٢ هو الانسب، وقد اعتمدت مقاس صفحات A4 مناسب للقراءة من خلال شاشة الحاسب او عند طباعة المحتوى على الاوراق بوضعية افقية معروفه باسم landscape.

كافة الشفرات البرمجية المستخدمة بالكتاب توجد بملف منفرد اسمه BECOME AN XCODER - source code.txt وهو يحتوي ماورد من أكواد حسب كل فصل. لقد تم البدأ بهذا العمل والانتهاء منه خلال شهر ونيف من العام ٢٠٠٦، وهو مجهود فردي –فلا تبخل على بالتصويب– آمل ان تستمع بقراءتة والاستفادة منه كما استمتعت واستفدت منه.

مازن الرمال

جدة – ٢٠٠٦